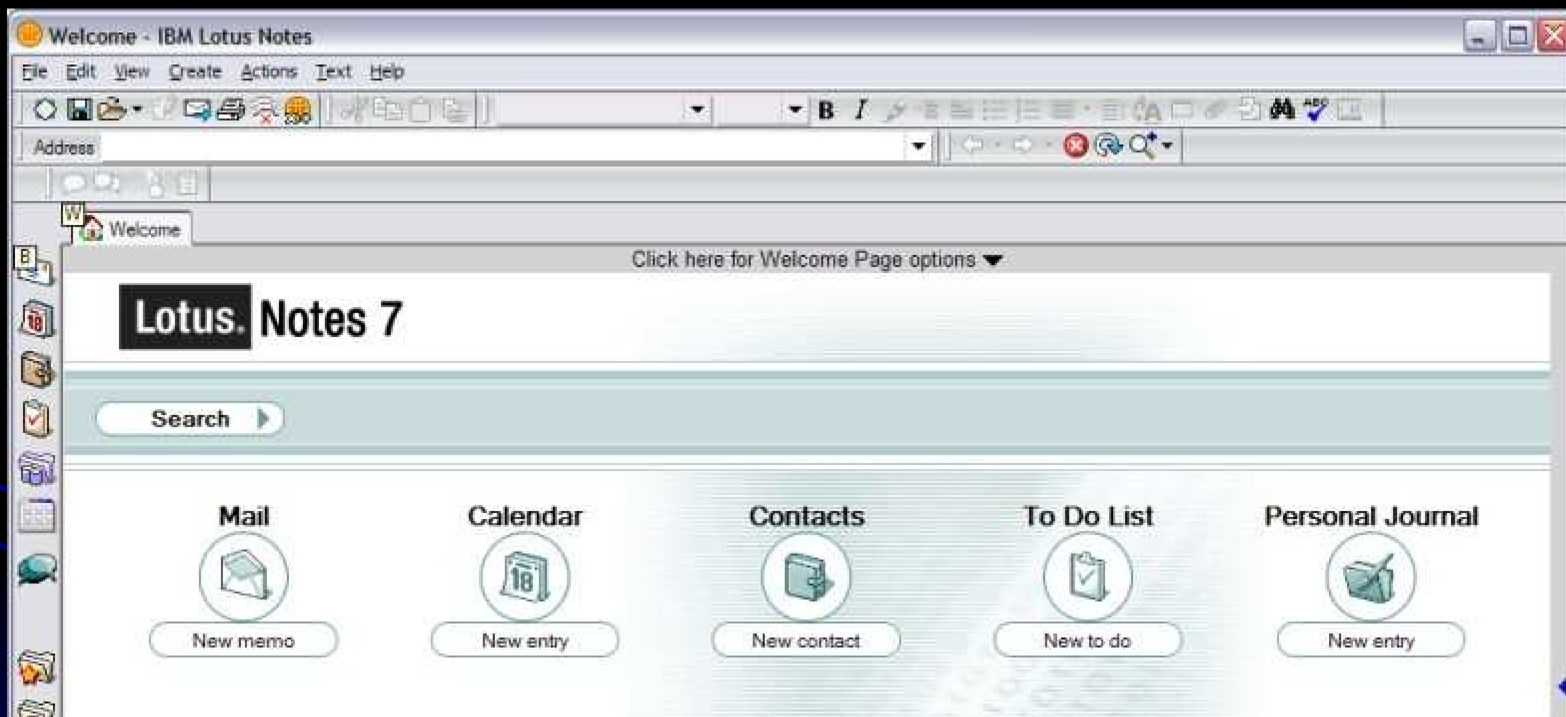
# Mitigating Dictionary Attacks on Password-Protected Local Storage

Ran Canetti, Shai Halevi, Michael Steiner
IBM T. J. Watson Research Center

http://eprint.iacr.org/2006/276

# Motivation

- This is how we start our day at IBM:




- What happens when I type in my password?

# Encrypted Local File

- Notes' startup does not rely on network
  - Configuration/credentials stored on local disk
  - Credentials are encrypted
- Upon startup
  - Derive key from password
  - Use it to unlock the credentials
  - Then use credentials for everything else

# What if I lose my laptop?

- No worries, my credentials are protected
  - My top secret password (`sh@1`) to the rescue
- What about users with weak passwords?
  - Attacker can mount off-line dictionary attack:
  - For each password in the dictionary
    - Derive a key from the password
    - Check if it decrypts the file
- Is it possible to protect against it?
  - **Without connectivity or secure hardware?**

# Other solutions (out of scope)

- Relying on "secure hardware":
  - Store secrets in secure hardware, use password to unlock hardware
    - Restrict password-guessing attacks
- Relying on the network:
  - Store secrets on server, use password to authenticate to server, then get secrets
    - Mitigate on-line attacks
- **This work: what if you cannot do either?**

# Key-derivation from passwords

- Common practice: use salt and a deliberately-slow key derivation function
  - E.g., key = $SHA1^{65536}$(salt | passowrd)
    - Different salt values for different users, salt is stored on disk.
- Linear slow-down for the attacker
  - But 65536 must grow as computers get faster
- Can we do better?

# A Different Approach

- A key-derivation protocol (user↔laptop)
  - User does more than just providing password
  - Using "human-only solvable puzzles"
    - People can solve these puzzles
    - Computers cannot
- User enter password, solves puzzles
- Key is derived from both password and the solutions to these puzzles

# Puzzles

# CAPTCHAs [Na96, vABHL03]

- Example: What's written here? 
- Can generate automatically with solution
  - Without any secret information
- People can usually solve them
- Solving them automatically is beyond the state of the art

# CAPTCHAs & pwds [PS02]

- Limit on-line attacks in client-server setting
  - Server generates CAPTCHAs + answers, sends to user
  - User sends back solutions, server checks
  - Then run pwd-based key-exchange protocol
- Not applicable in our setting
  - Where to store the solutions?

# Inkblot authentication [SS04]

- A different type of puzzles: one user's solution is unpredictable by other users

What do you see in  this picture?

- Solutions used to generate a strong pwd

# Is Inkblots the answer?

- Nice work if you can get it…
  - Unpredictability from other people a tall order
  - Need “many bits” in each puzzle
  - People should remember their answers
- How do we know that they are really unpredictable?
  - Maybe with some demographic information they become predictable…

# This Work

- A more robust protocol
- Same protection when puzzles are inkblots
- But strong protection even if solutions are predictable by other people
  - Can use CAPTCHAs (or in between)

# Our protocol

- Many puzzles are stored on disk ($z_1 \ldots z_n$)
- User's password is used to select a few
  - $\langle i_1, \ldots i_\ell \rangle \leftarrow$ **Expand**(salt, password)
- User is asked to solve these few puzzles
  - Solutions are $s_1, \ldots, s_\ell$
- Key is derived from password**+solutions**
  - Key $\leftarrow$ **Extract**(salt, $s_1, \ldots, s_\ell$, password)
- **Goal: attacker must solve MANY puzzles to find the key**

# An example

- Store $2^{20}$ CAPTCHAs (fit on one DVD-R)
- User needs to solve eight CAPTCHAs
  - $<i_1,\ldots i_8> \leftarrow$ HMAC-SHA1$_{salt1}$(pwd)
  - Each index is 20-bit long
- key $\leftarrow$ HMAC-SHA1$_{salt2}$($s_1,\ldots,s_8$, pwd)
- An attacker that solves 10,000 CAPTCHAs has $<$ 1% chance of hitting four of the eight CAPTCHAs that the honest user uses

# Properties of puzzles

- Automatically-generated problems
  - $z \leftarrow G(aux)$, G is randomized
  - aux can be user-supplied input (family pics?)
- Consistently solved by each human user
  - $s \leftarrow H(z)$, consistent across time*
  - Different users need not agree on an answer
  - But answers need not be unpredictable
- Hard to solve for a machine

* Can we use fuzzy extractors to correct a few errors?

# "Human-only solvable puzzles"

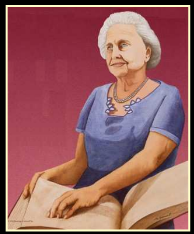
- Fairly weak requirements
- Need not be CAPTCHAs
  - Don't need to generate puzzle+solution
  - Not necessarily one right solution
- Need not be Inkblot
  - One user's solution not necessarily unpredictable by other users
- Can be many things in between

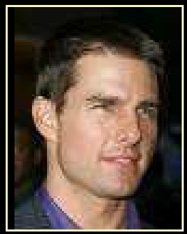
# Toy Examples

- Rank these people by coolness


Andrew Yao


Um Kulthum


Helen Keller


Tom Cruise

- Which of these pictures doesn’t belong?

# Hardness of puzzles

- What we need: hard to distinguish the "real solution" from a "random solution"
- (G,H) is μ-hard if there exists distribution R with μ bits of min-entropy such that
  - $z \leftarrow G()$ is a random puzzle
  - $s \leftarrow H(z)$ is the right solution
  - $s' \xleftarrow{\$} R(z)$ is a random solution
  - Attacker (PPT) cannot tell (z,s) from (z,s')

# Challenge: design good puzzles

- Need "many bits of hardness" for a construction to be useful
  - Else user is bothered with many puzzles
- Aside: must we store puzzles on disk?
  - Use r ß *f*(salt,pwd) as randomness to generate the puzzles?
    - Say, *f* is a random oracle
  - Puzzles must be hard even if attacker knows the randomness

# Security Analysis

# Adversarial model for protocol

- Attacker: not just PPT TM
  - Can also get help from people
- Protocol has access to human help, why not the attacker?
- This is a realistic attack
  - Used against deployed CAPTCHA systems
  - Attackers ship CAPTCHAs to their own web-sites, ask their visitors to solve them

# Modeling “human attackers”

- People can do many things
  - Outside the model: invite target to dinner, get her to disclose her password
- We assume: attacker only uses humans as puzzle-solvers
  - Attacker has oracle access to H
    - Or a “noisy version” of it (?)
  - Makes analysis possible
  - Keep in mind that it is not entirely realistic

# Formal adversarial model

- Attacker: efficient automated program (PPT TM) with puzzle-solving oracle
- Resources: time, number of queries
  - E.g., polynomial-time, sub-linear # of queries
- Goal: distinguish key from random

# Notions of security (1)

- Indistinguishability ([BR93]-style)
  - Attacker gets key, puzzles, salt, needs to decide if key is real or random
  - Will focus on this notion in this talk
- Bound attacker's advantage in terms of:
  - Parameters (*n* puzzles on disk, user solves $\ell$)
  - Size of password dictionary (|D|=d)
  - Number of oracle queries ($q \ll n$ queries)
  - Hardness of puzzles vs. key-length

# Notions of security (2)

- UC: define "ideal functionality" that only allows a limited number of password guesses:
  - Parameters: D: dictionary, p: #-of-pwd-guesses
  - Init(pw,aux) from user U
    - store pw, generate and store random key
    - Send aux to adversary
    - Check that pw$\in$D, else give it to adversary
  - Recover(pw') from anyone
    - If pw'=pw then return key to U
  - Password(pw') from adversary
    - If already made p such queries then ignore
    - Else if pw'=pw return key to adversary

# A “generic” attack

- An attack with complexity ~ $|D| \cdot 2^{\mu\ell}$
  - Attacker does not solve puzzles
  - Works even in the random-oracle model
- Assume: given a puzzle *z*, attacker can generate a list of $2^{\mu}$ potential solutions
  - The right solution is in the list
  - But the attacker does not know where
- This is consistent with a $\mu$-hard puzzle
  - And realistic attackers often have this ability

# The attack setup

- Attacker gets ($z_1$, $z_2$, …, $z_n$), salt (if any), and an alleged *m*-bit key $k^*$
  - Can make $\leq q$ queries to puzzle-solving oracle
- Needs to distinguish between:
  - "Random": $k^*$ is random,
  - "Real": $k^* \leftarrow$ **Extract**(salt, $s_{i1}, \ldots, s_{i\ell}$, pwd), where $\langle i_1, \ldots i_\ell \rangle \leftarrow$ **Expand**(salt, pwd) and $s_i$ is the right solution for $z_i$

# Generic attack, phase 1

- For each $p \in D$
  - Compute $<i_1,\ldots i_\ell> \leftarrow$ **Expand**(salt, p)
  - Generate $2^\mu$ solutions for each $z_{ij}$
    - For a total of $2^{\mu\ell}$ solution-vectors
  - Keep only those solution-vectors with **Extract**(salt, $s_1,\ldots,s_\ell$, p)=$k^*$
  - These are the "consistent vectors"
- So far, didn't make any oracle queries
  - If $m \gg \mu\ell$, can already distinguish

# Generic attack, phase 2

- Query the puzzle-solver upto *q* times
- Purge vectors $(s_1, \ldots, s_\ell, p)$ for which any of the solutions is not the right one
- Choose queries to maximize mutual-info between the answer and your decision
  - Or use some greedy strategy

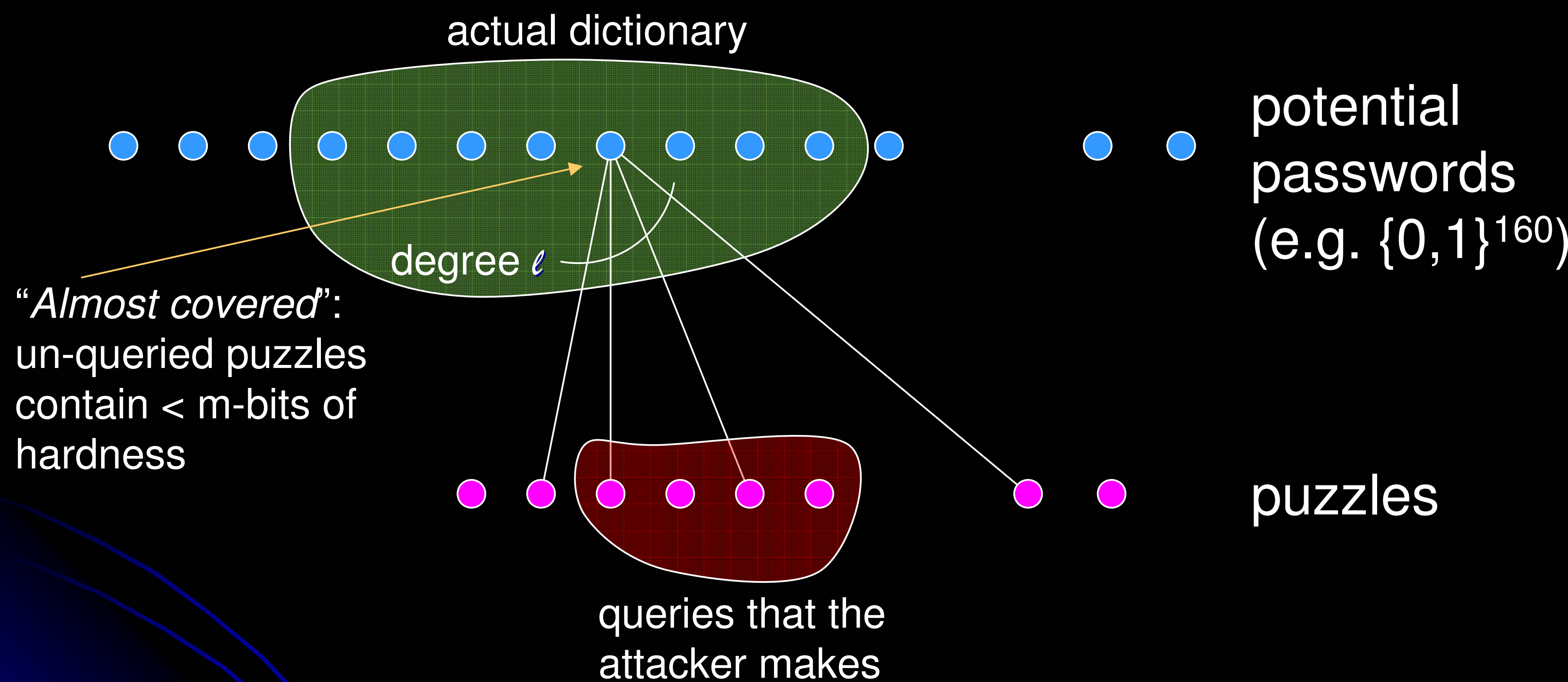
# Generic attack, phase 3

- Count remaining solution-vectors that are consistent with each password
- Maximum-likelihood: are these numbers more likely for “real” or “random”?
- The point: once we ask on “too many puzzles” for a password, we expect to have zero remaining consistent solutions in the “random” case

# The moral

- Attacker must query its oracle on many puzzles that are mapped to right pwd
- We prove: not just a feature of this attack
  - As long as remaining un-queried puzzles have more pseudo-entropy than key-length, key is secure
- Many many details/open problems
  - Some examples next, more on ePrint

# The function Expand




- Goal: attacker's queries "almost cover" only small fraction of the actual dictionary

# The function Expand (2)

- $\forall$ large enough D, $\forall$ $|Q|=q$ ($q \ll n$),

  a-cover(Q) is a very small fraction of D
  - If we were talking about cover(Q): the neighbor-set of any large subset of D contains more that |Q| neighbors (expansion)
  - Since we want a-cover(Q): same holds even when dropping many edges
    - As long as the degree remains > $m/\mu$
    - Similar to fault-tolerant expansion

# Constructing Expand

- Huge pwd-universe ($\{0,1\}^{160}$)
  => no deterministic construction
  - Deterministic construction for small D? (open?)
  - Randomized construction?
- Expand as a truly random function
  - Standard analysis using Chernoff
  - Ugly bound, but useful in specific cases
- Expand as $\ell n$-wise independent
  - Use n-th moment inequality

# Constructing Expand (2)

- Can we do better?
- Speculation: $\ell$ independent random linear maps over GF(2) work well
  - Old result of Alon et al. ("linear hashing yields small buckets"): a linear map over GF(2) works well for $q=\ell=1$, and $|D|=n$

# The function Extract

- Extracts m-bit key from puzzle solutions
- Key is pseudo-random as long as >m bits of pseudo-min-entropy are left in un-queried puzzles
- Strong randomness extractor is sufficient
  - From m*>m bits of min-entropy, extracts a key that is $\delta$ away from uniform m-bit sting
  - $\ell$* - number of puzzles needed to get m* bits of pseudo-min-entropy

# Security Statement

- Assuming puzzles are $\mu$-hard (and fixing parameters n, $\ell$, m, D, $\ell^*$, $\delta$), an attacker makeing q queries has advantage at most

$$\text{a-cover}(q,D) + \delta \cdot \binom{\ell}{\ell^*} + \text{negligible}$$

Caveat Emptor

# Non-malleability of puzzles

- Current analysis allows the attacker to query its humans only on puzzles that are stored on the disk
- To remove this restriction, puzzles need to be non-malleable
- Not clear how to define/achieve non-mal
  - Even if we don't care about human-solvable, e.g., non-malleable OWFs, PRGs, …

# Some open problems

- Design good puzzle systems
- Design of **Expand** function
  - Should be good (fault-tolerant) expander
- Better protocols (< storage, > security, etc.)
- (Non)-malleability of puzzles
- Better modeling of the attacker
- Better UC analysis

Thank you