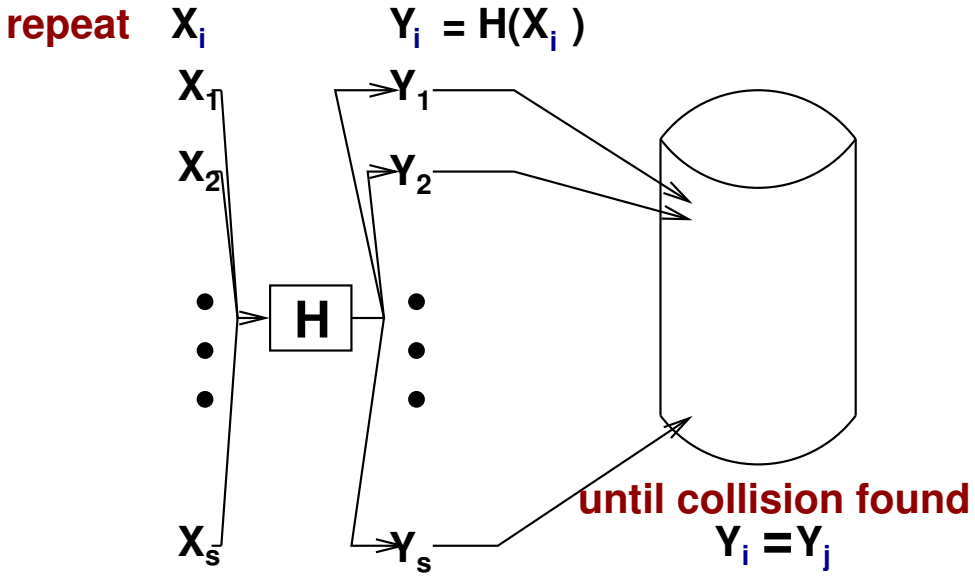
3: Generic Attacks

- consider a really good n-bit hash function
- we know we can find collisions in time $\Theta(2^{n/2})$



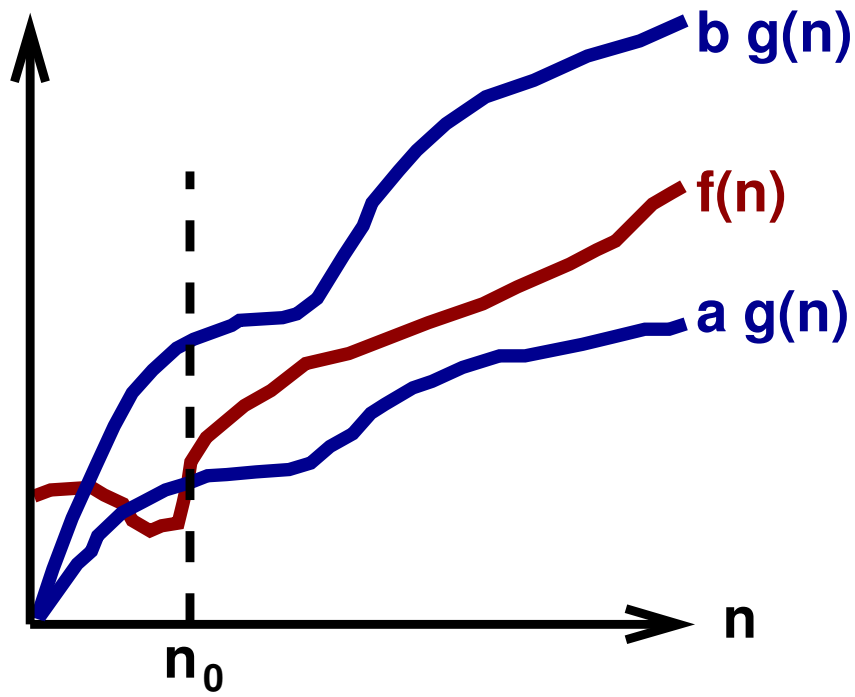
but this requires $\Theta(2^{n/2})$ units of memory.

Our goals:

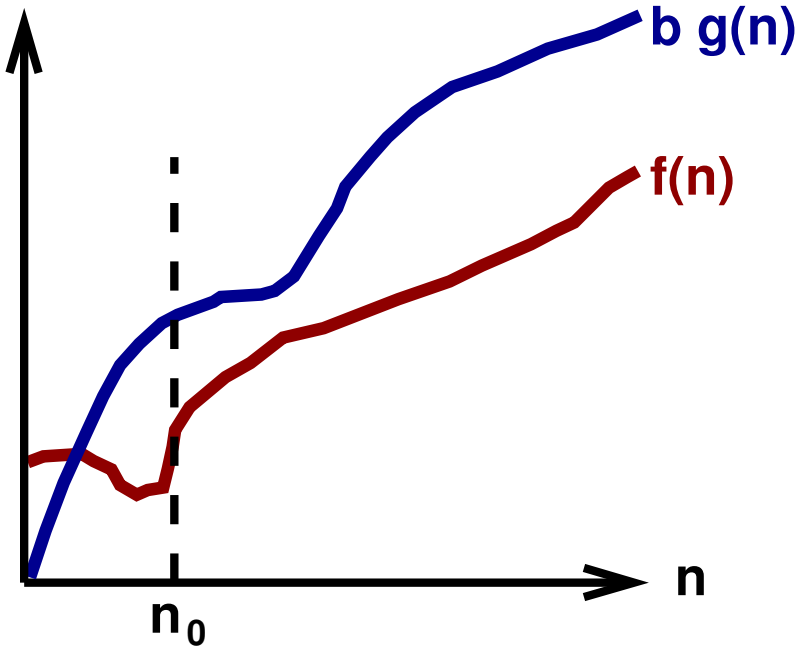
1. same time, less memory
2. on K machines in parallel: $\Theta(K)$-times faster ("linear speed-up")
3. also application to preimage-search ("password-cracking")

Theta-Notation: $\Theta(g(n)) =$

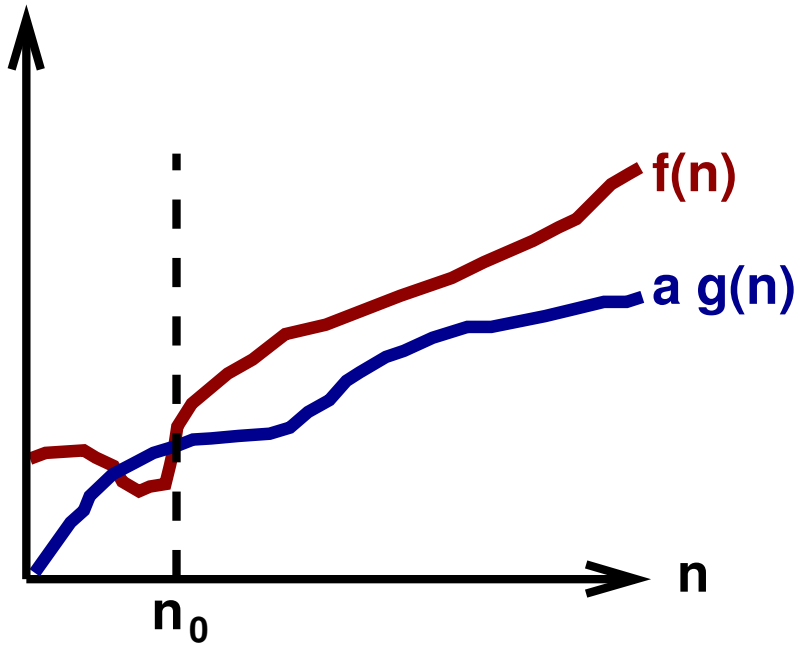
$$\left\{ \mathbf{f(n)} \mid \exists a, b, n_0 : \forall n > n_0 : \mathbf{0 < ag(n) \leq f(n) \leq bg(n)} \right\}$$



upper bound: $O(g(n))$



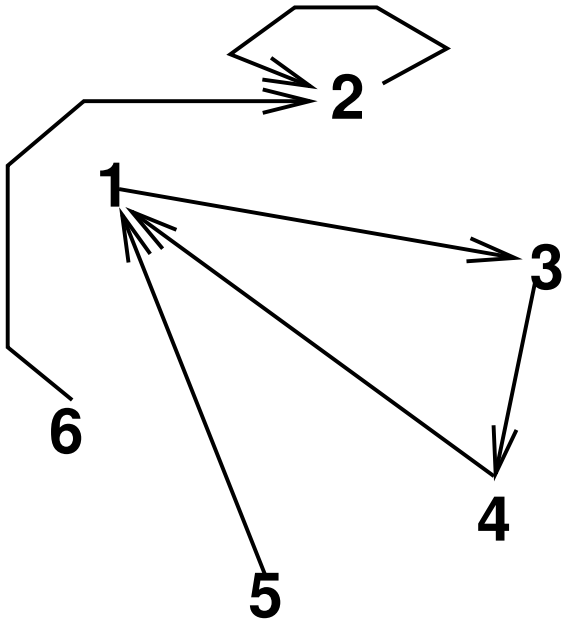
lower bound: $\Omega(g(n))$



$$O(g(n)) = \left\{ \mathbf{f(n)} \mid \exists b, n_0 : \forall n > n_0 : \mathbf{0} < \mathbf{f(n)} \leq \mathbf{bg(n)} \right\}$$

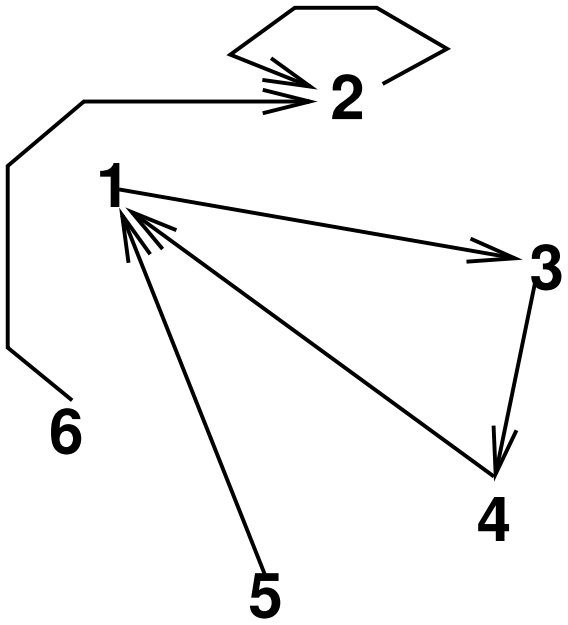
$$\Omega(g(n)) = \left\{ \mathbf{f(n)} \mid \exists a, n_0 : \forall n > n_0 : \mathbf{0} < \mathbf{ag(n)} \leq \mathbf{f(n)} \right\}$$

Random Functions



- $f : \{1, \ldots, 2^n\} \to \{1, \ldots, 2^n\}$
- small n: complete truth table
- large n:
  - simulation (“lazy evaluation”)
  - or formal model (“random oracle”).

Iterating Random Functions



- random "starting point" $x_0 \in \{1, \ldots, 2^n\}$
- "iteration" $x_i = f(x_{i-1})$
- **when will the x_i repeat?**
  - pre-period $\mu \in \Theta(2^{n/2})$,
  - period $\lambda \in \Theta(2^{n/2})$.
- with overwhelming probability (large n):
  - x_0 in pre-period
  - and thus collision $i \neq j$: $x_i = x_j$

3.1: Cycle Finding Algorithms

Objects x_i

- assignment
- test for equality
- compute $f(x_i)$

Applications of cycle finding

- collisions for hash functions
- test the statistical quality of random number generators
- factorize / compute the discrete logarithm
- compute infinite loops in complex programs
- …

Applying Cycle Finding Algorithms to Collision Search

1. start with x_0; iterate until in period;
2. compute length of pre-period and length λ of period;
3. if length of pre-period > 0:

   search for smallest $\mathbf{j}$ with $x_{\mathbf{j}} = x_{\mathbf{j}+\lambda}$.

   observe
   - $x_{\mathbf{j}-1}$ is the last point in the pre-period,
   - $x_{\mathbf{j}}$ is the first in the period

   thus $X_{\mathbf{j}-1}$ and $x_{\mathbf{j}+\lambda-1}$ collide

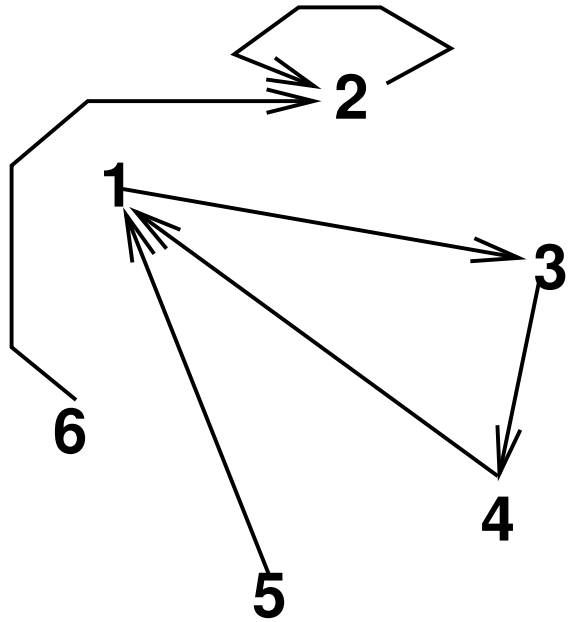
Algorithms with constant (!) memory and expected time $\Theta(2^{n/2})$:

- Floyd Cycle Finding (very well-known) or
- Brent Cycle Finding (on the average, a bit more efficient).

Brent Cycle Finding

start with x_0;
iterate until in period:

turtle $:= x_0$;
hare $:= f(x_0)$;
$\lambda := 1$; $p := 1$;
while turtle $\neq$ hare
  if $p = \lambda$ **then**
    turtle $:=$ hare;
    $p := 2p$;
    $\lambda := 0$;
  hare $:= f$(hare);
  $\lambda := \lambda + 1$;
output (λ, hare);

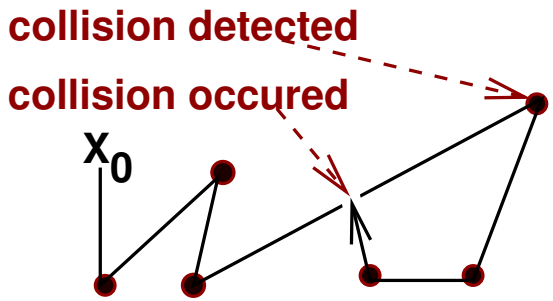


Time-Memory Trade-offs

Additional operations on objects

- store objects in a (small!) table T and
- check, whether an object is in T (and if yes, at which position)

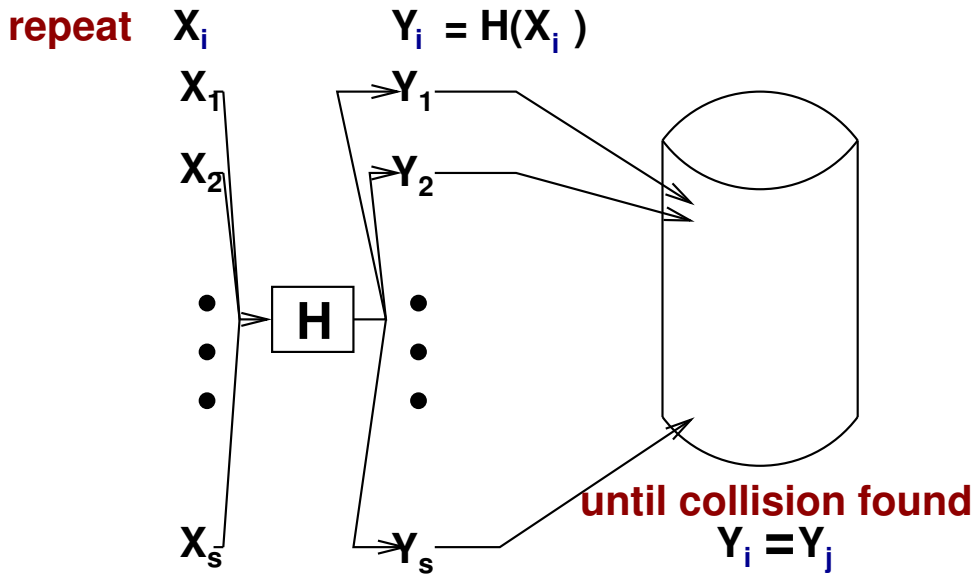
1. extreme case: table of size $\Theta(2^{n/2})$ (naive algorithm)
2. extreme case: Brent ("table" of size 1, for the turtle)
3. trade-off: 2^n points, $2^n/Z$ *Distinguished Points*:
   - e.g., set $Z = 2^z$; define all points $X_i \in \{1, \ldots, 2^n\}$ whose binary representation end with 0^z to be *Distinguished Points*
   - store only DPs in table
   - look up DPs, only



3.2: Parallel Collision Search

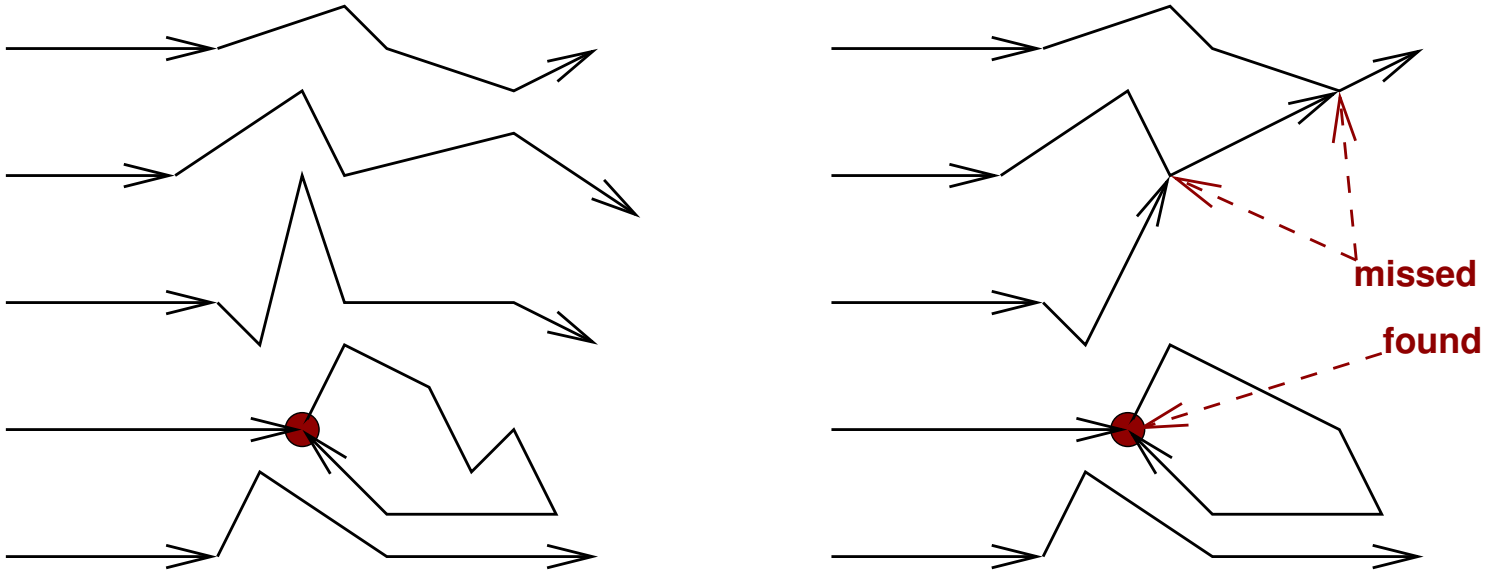
Given K machines, how fast can we find collisions?

- naive memory-inefficient: $\Theta(2^{n/2}/K)$ time (= calls to f)
  - **linear speed-up** (**wow!**),
  - but only in a theoretical model (neglecting communication)



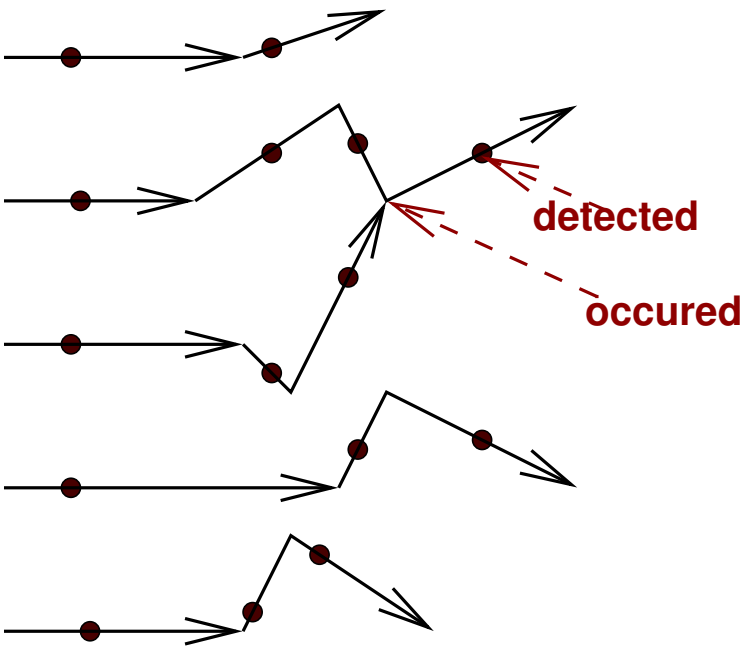
Memory-Efficient Methods

- Independent cycle finding (Brent, Floyd, Distinguished Points) on K parallel machines:



  - $c \cdot 2^{n/2}$ steps $\rightarrow$ expected number of collisions $\Theta(c^2)$.
  - K different starting points – we would expect memory-efficient collisions in time $\Theta(\sqrt{2^n/K})$.
  - Even if this would work as good as we naively expect, **the speed-up is only $\sqrt{K}$**!
  - Note that we actually **miss most collisions**!

Improved Parallel Collision Search



- K different starting points,
- store all Distinguished Points in the same database:
  - storage size and communication overhead are reasonably small if Z is large enough
  - the probability to miss collisions is small if Z is not too large
- run time $\Theta(2^{n/2}/K)$ (**linear speed-up!**)

$\Rightarrow$ **The preferred method in practice!**

- How to find the actual collision?

3.3: Preimages

- function $H : \mathcal{X} \to \mathcal{Y}$ (any (good) hash function)
- given $Y = H(X')$ find X with $H(X) = Y$
- sometimes, we care about $X = X'$, sometimes we do not

- a typical application: known plaintext attacks
  given $Y = E_X(\text{Plaintext})$, find X
- another typical application: password-cracking secret password X,
  known: (Username, $H(X)$)
- our algorithms are application-independent

More on Password-Cracking

- function $H : \mathcal{X} \to \mathcal{Y}$ (may depend on "salt")
- given $Y = H(X')$, find X with $H(X) = Y$

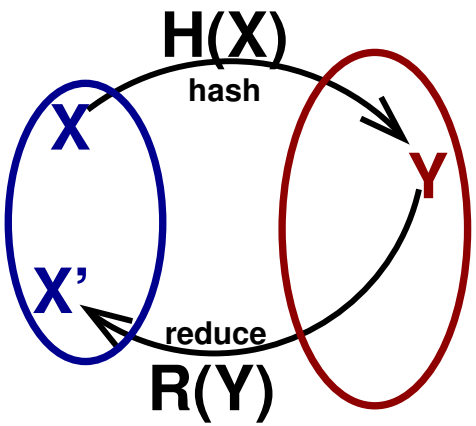
- password-space $\mathcal{X}$
- with $M \leq |\mathcal{X}|$ the number of "plausible" passwords (e.g., from a dictionary)
- 2^n possible hash values $H(\cdot)$, $M \ll 2^n$.
- naive approach: time $\Theta(M)$, essentially no memory, easy parallelizable
- **great!** we cannot improve on this, can we?

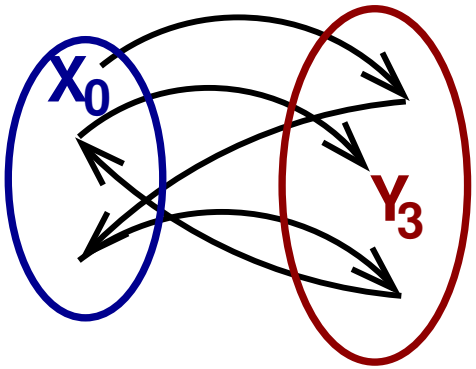
Quickly cracking many passwords

- given $Y_1 = H(X'_1), \ldots, Y_L = H(X'_L)$,
- search for X_i with $H(X_i) = Y_i$ for as many Y_i as possible
- "Table Look-Up":
  - preparation time: $\Theta(M)$,
  - storage: $\Theta(M)$,
  - time for each Y_i: $\Theta(1)$.

  due to memory constraints often infeasible
- can we do something like that with less memory?

“Time-Memory Trade-off” (Hellman, 1980)





- function $R : \mathcal{Y} \to \mathcal{X}$ (“reduction”), mapping any hash value into the password-space

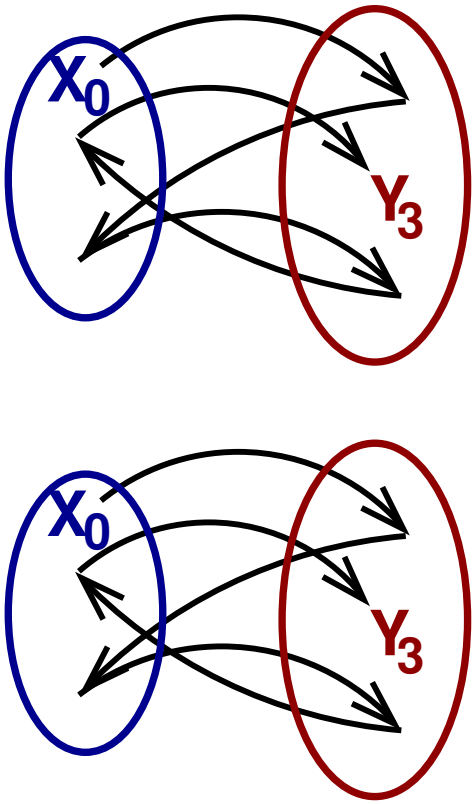
  but R is not the inverse function of H!

- We use R and H to build “chains”:

$$\begin{aligned} X_0 &:= \text{start}; & Y_1 &:= H(X_0); \\ X_1 &:= R(Y_0); & Y_2 &:= H(X_1); \\ &\vdots \\ X_{L-1} &:= R(Y_0); & Y_2 &:= H(X_1); \end{aligned}$$

- the “trick” to save storage is to only store the endpoint Y_L

The “Chains”



- assume, during preparation time we build
  - K chains
  - each of length L.
- then we did
  - consider at most (!) KL passwords
  - using K units of storage
- let $Y = H(X)$ be given; if X is one of the passwords considered, then
  - we can compute X in time $\Theta(L)$
  - (how?)

This Time, Collisions are bad for the Attacker!

- ideally: storage and time (per pw) each at $K = L = \Theta(2^{n/2})$
  - **but:** too many collisions between different chains
  - we only deal with $\ll KL$ *different* passwords in time $\Theta(2^n) = KL$

- so we reduce our parameter sizes:
  - smaller values $K = L = \Theta(2^{n/3})$
  - storage and time (per pw) each $\Theta(2^{n/3})$
  - **good:** $\Theta(1)$ collisions – we consider $\approx KL$ different passwords
  - **but** $KL = \Theta(2^{2n/3})$ is small – we would find one in $\Theta(2^{n/3})$ pws

- $T = \Theta(2^{n/3})$ different tables with different "reduction" functions R allows to cover almost all passwords
  - preparation time: $KLT = \Theta(2^n)$,
  - storage: $KT = \Theta(2^{2n/3})$,
  - time per password: $LT = \Theta(2^{2n/3})$.

Improvements and Defenses

Improvements:

- There are some ways to improve the constants hidden in $\Theta(\cdot)$:
  - longer chains, abort when collision found (using Dist. Points),
  - or “rainbow-tables”.
- No known method for an asymptotical improvement!

Defenses:

- Use a random “salt” S.
  I.e., store (Username, S, $H(S||X)$).
  (Why does this defend against TMTO-attacks?)
- Use a intentionally slow and resource-demanding H, instead of a fast cryptographic hash.