Hash Functions — Generic Attacks

Orr Dunkelman

Computer Science Department

14 March, 2012

Outline

Basic Inversion Attacks

- A function $f : \{1, 2, \ldots, N\} \mapsto \{1, 2, \ldots, N\}$ is fixed.
- A value $y = f(x)$ is given to the attacker who needs to find x.
- This problem can model finding the inverse function of a compression function or a hash function.
- This problem can model finding keys of an encryption function.
- For example, $f(k) = E_k(P)$ for some pre-determined plaintext P.
- Or $h(\text{input})$ for hash functions.

Basic Inversion Attacks (cont.)

Simple approaches for solution:

- Exhaustive search — the attacker computes for each i the value of $f(i)$ and stops once $y = f(i)$.
- Table attack/Dictionary attack — the attacker precomputes once all $f(i)$, and stores in a table $(f(i), i)$ sorted according to $f(i)$.

- Exhaustive search — Precomputation $= 0$; Memory $= 0$; Time $= N$.
- Table — Precomputation $= N$; Memory $= N$; Time $= 1$.

Some Variants Exhaustive Search

- If P CPUs are available, we can let each CPU run over $1/P$ of the search space.

Time $= N/P$ (in real time).

- Sometimes, it is possible to evaluate $f(\cdot)$ on several points simultaneously (bit-slicing, same subkey in the first round, etc.)

Does not affect asymptotic time, but actual time.

- Sometimes, it is possible to partially-evaluate $f(\cdot)$, and only if the partial evaluation succeeds, compute the full evaluation.

Does not affect asymptotic time, but actual time.

- For a specific $f(\cdot)$, it is usually more efficient to build dedicated hardware (FPGA/ASIC).

Saves on the money/time ratio (does not affect asymptotic).

Diffie and Hellman's DES Machine

- Shortly after the introduction of DES, Diffie and Hellman analyzed the 56-bit key length.
- Machine with 1,000,000 chips (in 64 racks), each tests a DES key in a microsecond.
- Connecting it all (and taking some overhead), their machine could find a DES key every half a day on average for 20,000,000$.
- If you run the machine for 5 years, the cost of finding a key is expected to be 5000$.

For more information: W. Diffie, M. Hellman, *Exhaustive cryptanalysis of the NBS Data Encryption Standard*, Computer, vol. 10, no. 6, pp. 74–84, June 1977.

Exhaustive Search Example — DES Challenges

- In 1997 RSA Labs started a DES challenge, they published a plaintext and a ciphertext, and offered a prize for the first one finding the key.
- The DESCALL project was used to solve the first challenge in a distributed manner (90 days).
- In 1998, the second DES challenge was launched.
- distributed.net project found the key in 39 days.
- The third challenge (and last) was cracked using the DES Cracker (by EFF).
- 22 hours to find a random key of 56-bit (full exhaustive search was expected to take 56 hours).

Exhaustive Search Example — DES Cracker

- DES cracker consisted of 1,536 custom-designed ASIC chips at a cost of material of around 250,000$ and could search 88 billion keys per second.
- That is more than 2^{36} keys per second.
- A full exhaustive search requires about 819,000 seconds (slightly less than 9.5 days).
- Actually, the majority of the cost is the design cost and fabrication of the first unit.
- A second machine would be much cheaper.
- Actually, today, for the same price, one should expect 256 more computational power for the same cost (or the same computational power for slightly less than 1000$) following Moore's law.

Exhaustive Search Example — DES Cracker (cont)

- DES Cracker was used in the third challenge, and found the key in 56 hours.
- In the amended third challenge (issued two weeks later), the DES cracker was integrated into the Distributed.Net project. This challenge was solved in about 22 hours.
- Conclusion: 56-bit key is not secure (1997).
- Conclusion 2: Today, 64-bit key is not secure.
- Conclusion 3: For real security, move to 80-bit security.

COPACOBANA

- COPACOBANA is a board with 120 slots for FPGAs.
- Using a hardware optimized implementation of DES, an (old) FPGA can check one key every 1.84 ns.
- Put 120 such FPGAs together, and you can search 65.3 billion keys per second.
- In total, a search space of 56-bit DES key requires about 12.7 days to fully exhaust.
- Cost estimates: 120 (old) FPGAs at less than 10,000 Euros.

More info: T. Güneysu et al. *Cryptanalysis with COPACOBANA*, IEEE Transactions on Computers, vol. 57, no. 11, November 2008.

Sony PlayStation

- In a sequence of recent attacks, the need for a large computing power has raised.
- The attacks (specifically designated at MD5-based certificates) required computing many MD5 executions quickly.
- Sony PlayStation has 7 processing units. Which means you can run the code 7 times in parallel (at a slightly slower clock rate).
- By explicit use of bit-slicing techniques and the Sony PlayStation 3 architecture, an amount of 175 million MD5 computations per second per machine could be reached (cost of machine — 400$).

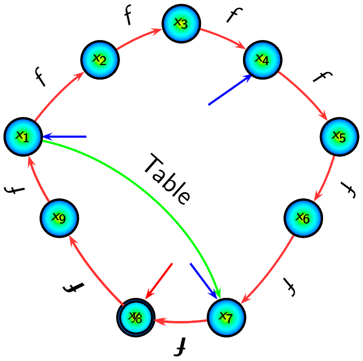
Graphics Processing Units (GPU)

- There is actually nothing that amazing in PlayStations.
- The PlayStations were chosen as the ratio of MD5 computations in second per $ was better than using many computers.
- One can improve this ratio using GPUs. On an ATI HD 4850 X2, it is possible to compute 1634 million MD5 computations per second (at 245$).

Hellman's Time-Memory Tradeoff Attack

- In [H80] Hellman suggested a method to have a tradeoff between the time and the memory complexities.

- Assume that f is a permutation, such that it has one huge cycle covering all values.
- Precomputation: pick at random point x_1, compute $x_{i+1} = f(x_i)$, and store the $\sqrt{N}$th values (i.e., $x_1, x_{\sqrt{N}+1}, x_{2\sqrt{N}+1}, \ldots$).
- Online phase: given y, compute $f^j(y)$ until a stored $x_{i\sqrt{N}}$ is encountered. Obtain $x_{(i-1)\sqrt{N}+1}$ from the table, and apply f to it, until y is obtained.



Hellman's Time-Memory Tradeoff Attack (cont.)

- The suggested attack has precomputation of $P = N$, storage of $M = \sqrt{N}$, and online time, $T = \sqrt{N}$.
- But this works only if f induces a single cycle!
- Actually if f is a permutation a similar attack works, which might require even less online computation or even less memory (or both).
- The idea is to store every $\sqrt{N}$th point on the cycle.
- Once the cycle is smaller than $\sqrt{N}$ there is no need to store any point on it, as you can start from y, and find its preimage (predecessor in the graph) in less than $\sqrt{N}$ operations.

Hellman's TM Attack on Random Functions

- First trial:
  - Precomputation: Take $m = \sqrt{N}$ starting points x^i and from each such point, generate a sequence of $\sqrt{N}$ values, and store the obtained end points (y^i, x^i).
  - Online phase: Given y, start computing f on it, until hitting one of the end points. Retrieve from the table the value of the corresponding start point x^i, and compute forward until $x = f^{-1}(y)$ is found.

$$\begin{tikzcd}
x^1 \arrow[r, "f"] & x^1_1 \arrow[r, "f"] & x^1_2 \arrow[r, "f"] & \ldots \arrow[r, "f"] & y^1 \\
x^2 \arrow[r, "f"] & x^2_1 \arrow[r, "f"] & x^2_2 \arrow[r, "f"] & \ldots \arrow[r, "f"] & y^2 \\
x^3 \arrow[r, "f"] & x^3_1 \arrow[r, "f"] & x^3_2 \arrow[r, "f"] & \ldots \arrow[r, "f"] & y^3 \\
 & & \vdots & & \\
x^m \arrow[r, "f"] & x^m_1 \arrow[r, "f"] & x^m_2 \arrow[r, "f"] & \ldots \arrow[r, "f"] & y^m
\end{tikzcd}$$

Hellman's TM Attack on Random Functions (cont.)

- The function f is random. Thus, there are collisions between the chains!
- From the collision, both chains "evolve" together, and thus cover the same nodes (values).
- Thus, the chains are expected to cover much less than N nodes.
- Adding more chains, will not solve the problem (each new chain will cover very few new nodes before a collision is found).
- Finally, because f is random, some nodes do not have predecessors (about $1/e$ of the space).

Hellman's TM Attack on Random Functions (cont.)

- Hellman solved the problem by using **different functions!**
- Let f_i be some small tweak of f, such that inverting f_i is like inverting f (for example $f_i(x) = f(x) \oplus i$).
- For each of the t functions f_i, pick m random starting points, and compute chains of length t.
- For each function, store the values $(end, start)$ in a table.
- In the online phase — try to compute $f_i^j(y)$ for every $i = 1, \ldots t$, and $j = 1, \ldots, t$, until one of the end points is found. Go to the corresponding start point, and find the predecessor of y.

Hellman's TM Attack on Random Functions (cont.)

- Preprocessing — N. Memory — t tables, of m blocks each, total of mt. Online time — t^2 applications of f, and t^2 table accesses. As we want to cover $O(N)$ values, we need $mt^2 \approx N$, i.e.,

$$TM^2 = N^2.$$

- A common point on the curve is $M = T = N^{2/3}$.
- Of course, if $mt > N$ or $t^2 > N$, then the attack is inferior to other generic attacks.
- There are some small technicalities concerning the false alarms (hitting an end point, even though the value is not covered by the chain), but most of the time it is OK.

Choosing the Function f

- Consider the case of a block cipher.
- When suggesting a function to invert, the function often picked is $f(K) = E_K(P)$ from some pre-determined plaintext P.
- When the block size is equal to the key size, this function has the "right" size.
- But what if the block size is not equal to the key size?
- If $|P| > |K|$, then $E_K(P)$ is longer than K, and a simple solution is to drop some bits of the output (beware of false alarms!).
- If $|P| < |K|$, one can define the function $f(K) = E_K(P_1)||E_K(P_2)$.

Choosing the Function f

- Actually the problem is that the attacks works for $f : \{1, 2, \ldots, N\} \mapsto \{1, 2, \ldots, N\}$, so we need to transform the function which we wish to invert $F(\cdot)$ to such a form.
- Hence, in most cases, we need to transform the "natural" output to be in the same domain as the input.
- This is extremely hard when the domains are not of the type "all strings of length ℓ" but have more complicated structures, e.g., "all passwords containing only characters in English".
- This is also a way to implement different f_i functions.

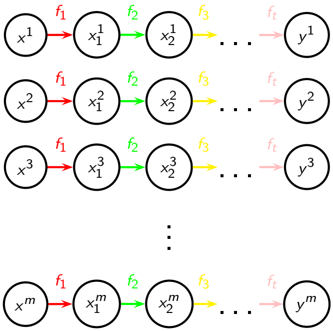
Reducing Storage Accesses using Distinguished Points

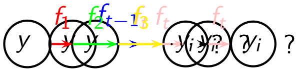
- Each table contains m pairs of (end point, start point).
- After each computation of f (or f_i), we need to access a table.
- Accessing large tables (especially if m is larger than the size of the RAM) takes time, sometimes greater than of actually computing f.
- A solution by Rivest, to use the concept of *distinguished points*.
- Instead of each chain ending always after t iterations of f_i, we let the chain continue until an easily identifiable point is achieved (e.g., $\log_2(t)$ least significant bits are 0).

Reducing Storage Accesses using Distinguished Points (cont.)

- On average, the same number of points is covered. But instead of t^2 table accesses, we can use only t of these (whenever a distinguished point is encountered).
- Note that the other parameters are the same! Specifically, the number of f invocations and/or memory size is the same.
- Really good for hardware acceleration and parallelization [SRQL02].

Rainbow Tables


$x^1 \xrightarrow{f_1} x_1^1 \xrightarrow{f_2} x_2^1 \xrightarrow{f_3} \ldots \xrightarrow{f_t} y^1$
$x^2 \xrightarrow{f_1} x_1^2 \xrightarrow{f_2} x_2^2 \xrightarrow{f_3} \ldots \xrightarrow{f_t} y^2$
$x^3 \xrightarrow{f_1} x_1^3 \xrightarrow{f_2} x_2^3 \xrightarrow{f_3} \ldots \xrightarrow{f_t} y^3$
$\vdots$
$x^m \xrightarrow{f_1} x_1^m \xrightarrow{f_2} x_2^m \xrightarrow{f_3} \ldots \xrightarrow{f_t} y^m$



- As noted before, it might be beneficial to reduce the number of accesses to the table.
- In [O03], Oechslin suggested the concept of rainbow tables, without the need of distinguished points.
- Instead of having t multiple tables (each with m starting points), we start with mt starting points.
- For each point x_i, we evaluate $y_i = f_t(f_{t-1}(\ldots f_2(f_1(x_i))\ldots))$, and store (y_i, x_i).
- In the online phase: given y, check $f_t(y)$, $f_t(f_{t-1}(y))$, ... as end

Rainbow Tables (Analysis)

- This method has the advantage of reducing false alarms, and it is claimed to achieve a curve $N^2 = 2TM^2$.
- This is partially true, but due to some technicalities, [BBS06] showed that rainbow tables are less favorable (mostly due to a larger memory block).

Analysis of Time-Memory Tradeoff Attacks

- The matter of estimating the success rate of a TMTO attack has received a great deal of attention.
- Coverage in one table was already rigorously analyzed by Hellman in 1980 (a table covers about 80% of its "size").
- Under the assumption the tables are independent of each other, the expected success rate is 55%.
- A lot of research was done on choosing the optimal values (see [BPV98,SRQL02,KM96,KM99] for more details).
- Then the rainbow table succeeds to offer more than 99% coverage.
- Why?

Analysis of Time-Memory Tradeoff Attacks

- The main problem is the interaction between the different f_i's, which are actually relatively close to each other.
- This is especially meaningful, when considering the fact that there are many points without an ancestor (about $1/e$ of the points of the graph has $d_{in} = 0$).
- Rainbow's 99% coverage stems from the fact that they target cases where there are many preimages (e.g., in hash function inversion).
- The problem was left open until [BBS06] which analyzed the success rate of generalized TMDTO algorithms in the *stateful random graph* model.
- In the stateful random graph model, the computation is applied on a state which has an hidden part.

How to Find the Collision

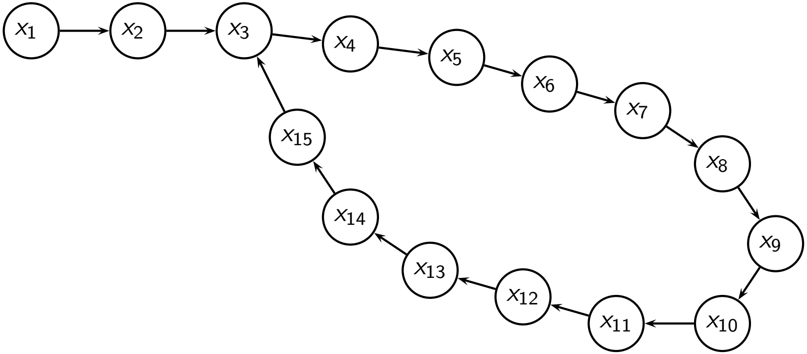
- The simplest algorithm is to pick $1.17 \cdot 2^{n/2}$ random values m_i, compute $h(m_i)$, and store them in a hash table, until a collision is found.
- Time complexity: $1.17 \cdot 2^{n/2}$ invocations of $h(\cdot) +$ $1.17 \cdot 2^{n/2}$ memory accesses.
- Memory complexity: $1.17 \cdot 2^{n/2}$ cells.

Reducing Memory Usage

- Consider the random function $f : \{0,1\}^n \mapsto \{0,1\}^n$ as a directed graph:
  - Let $V = \{0,1\}^n$ (i.e., each node has a label of length n).
  - and $(x,y) \in E$ if $f(x) = y$.
- A collision in $f(\cdot)$ can be views as two edges (x_1, y) and (x_2, y).

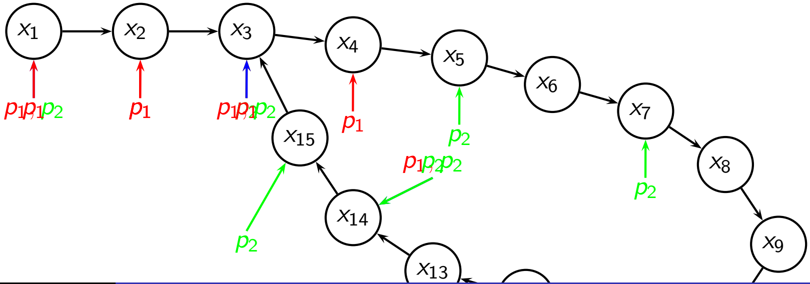
Cycle Finding

- Start from a random node x_1, and compute iteratively $x_{i+1} = f(x_i)$.
- After about $\sqrt{2^n}$ steps, you expect to enter a cycle.
- The entry point (unless it is back to x_1) suggests a cycle.



Floyd's Cycle Finding Algorithm

- Start with two pointers p_1, p_2 initialized both to x_1.
- p_1 is incremented each time by 1 position $p_1 \leftarrow f(p_1)$, and p_2 is incremented each time by 2 positions $p_2 \leftarrow f(f(p_2))$ until they collide.
- At this point, set p_1 to x_1, and increment both pointers each time by 1 position, they will collide in the entry point to the cycle.



Analysis of Floyd's Cycle Finding Algorithm

- This method is also known as the ρ-method.
- Let the tail's ($x_1 \rightsquigarrow x_3$) length be ℓ, and let the cycle's length be r. Then if the two pointers collide after t steps:

$$t - \ell = 2t - \ell \bmod r \Rightarrow t \equiv 0 \bmod r$$

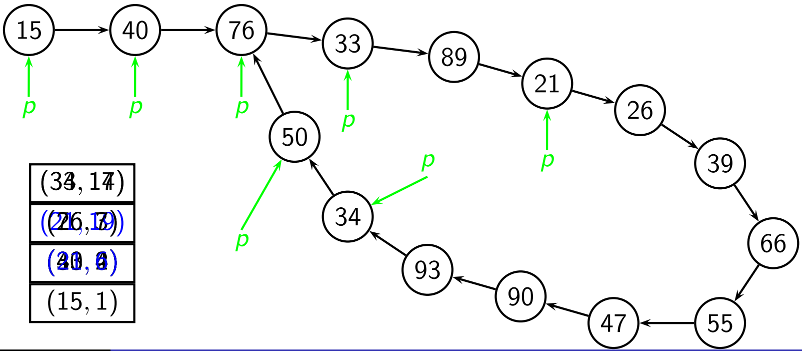
- Then, after ℓ more steps, the pointer p_2 is in position $2t + \ell$, which means, it did $2t$ steps inside the cycle, which means that it points to the entry point.
- The algorithm does not work when x_1 is the start of the cycle, or when the cycle is of length 1 (the former is easily solved by picking a different starting point, the latter offers a fixed-point).

Nivasch's Algorithm for Cycle Finding

- Floyd's algorithm may take up to $5r + 3\ell$ computations of f until the collision is located.
- The idea behind Nivasch's algorithm is to reduce the time significantly, while using a small amount of memory.
- It also uses only one pointer.

Nivasch's Algorithm for Cycle Finding (cont.)

- Initialize a pointer p to some random element $x_1 = r$.
- Initialize a stack to $(x_1, 1)$.
- Each step, compute $x_{i+1} = f(x_i)$.
- Pop from the stack all entries (x_j, j) for which $x_j > x_{i+1}$.
- Push $(x_{i+1}, i+1)$ to the stack.



Nivasch's Algorithm for Cycle Finding (cont.)

- Once the collision is detected, we can move to the phase of finding the actual collision.
- Note that the difference between the indices in the colliding entry is the length of the cycle.
- The memory consumption is about $O(n)$ for a space of 2^n values.
- It is also possible to use 2^k stacks (each corresponding to a different value of k LSBs of the value), thus detecting the colliding values faster (i.e., find the cycle length faster).
- Finally, this can be used to find the actual collision.