Birthday attack

A **birthday attack** is a type of cryptographic attack that exploits the mathematics behind the birthday problem in probability theory. This attack can be used to abuse communication between two or more parties. The attack depends on the higher likelihood of collisions found between random attack attempts and a fixed degree of permutations (pigeonholes), as described in the birthday problem/paradox.

Understanding the problem

As an example, consider the scenario in which a teacher with a class of 30 students asks for everybody's birthday, to determine whether any two students have the same birthday (corresponding to a hash collision as described below [for simplicity, ignore February 29]). Intuitively, this chance may seem small. If the teacher picked a specific day (say September 16), then the chance that at least one student was born on that specific day is $1-(364/365)^{30}$, about 7.9%. However, the probability that at least one student has the same birthday as *any* other student is around 70% (using the formula $1-365!/((365-n)!\cdot 365^{n})$ for n = 30).

Mathematics

Given a function f, the goal of the attack is to find two different inputs x_1, x_2 such that $f(x_1)=f(x_2)$. Such a pair x_1, x_2 is called a collision. The method used to find a collision is simply to evaluate the function f for different input values that may be chosen randomly or pseudorandomly until the same result is found more than once. Because of the birthday problem, this method can be rather efficient. Specifically, if a function $f(x)$ yields any of H different outputs with equal probability and H is sufficiently large, then we expect to obtain a pair of different arguments x_1 and x_2 with $f(x_1)=f(x_2)$ after evaluating the function for about $1.25\sqrt{H}$ different arguments on average.

We consider the following experiment. From a set of *H* values we choose *n* values uniformly at random thereby allowing repetitions. Let *p*(*n*; *H*) be the probability that during this experiment at least one value is chosen more than once. This probability can be approximated as

$$p(n;H)\approx 1-e^{-n(n-1)/(2H)}\approx 1-e^{-n^2/(2H)},$$

Let *n*(*p*; *H*) be the smallest number of values we have to choose, such that the probability for finding a collision is at least *p*. By inverting this expression above, we find the following approximation

$$n(p;H)\approx\sqrt{2H\ln\frac{1}{1-p}},$$

and assigning a 0.5 probability of collision we arrive at

$$n(0.5;H)\approx 1.1774\sqrt{H}.$$

Let *Q*(*H*) be the expected number of values we have to choose before finding the first collision. This number can be approximated by

$$Q(H)\approx\sqrt{\frac{\pi}{2}H}.$$

As an example, if a 64-bit hash is used, there are approximately 1.8×10^{19} different outputs. If these are all equally probable (the best case), then it would take 'only' approximately 5 billion attempts (5.1×10^{9}) to generate a collision using brute force. This value is called **birthday bound**[1] and for *n*-bit codes it could be computed as $2^{n/2}$. Other examples are as follows:

Bits	**Possible outputs (2 s.f.) (H)**	**Desired probability of random collision (2 s.f.) (p)**									
		$\mathbf{10^{-18}}$	$\mathbf{10^{-15}}$	$\mathbf{10^{-12}}$	$\mathbf{10^{-9}}$	$\mathbf{10^{-6}}$	**0.1%**	**1%**	**25%**	**50%**	**75%**
16	66,000	<2	<2	<2	<2	<2	11	36	190	300	430
32	4.3×10^{9}	<2	<2	<2	3	93	2900	9300	50,000	77,000	110,000
64	1.8×10^{19}	6	190	6100	190,000	6,100,000	1.9×10^{8}	6.1×10^{8}	3.3×10^{9}	5.1×10^{9}	7.2×10^{9}
128	3.4×10^{38}	2.6×10^{10}	8.2×10^{11}	2.6×10^{13}	8.2×10^{14}	2.6×10^{16}	8.3×10^{17}	2.6×10^{18}	1.4×10^{19}	2.2×10^{19}	3.1×10^{19}
256	1.2×10^{77}	4.8×10^{29}	1.5×10^{31}	4.8×10^{32}	1.5×10^{34}	4.8×10^{35}	1.5×10^{37}	4.8×10^{37}	2.6×10^{38}	4.0×10^{38}	5.7×10^{38}
384	3.9×10^{115}	8.9×10^{48}	2.8×10^{50}	8.9×10^{51}	2.8×10^{53}	8.9×10^{54}	2.8×10^{56}	8.9×10^{56}	4.8×10^{57}	7.4×10^{57}	1.0×10^{58}
512	1.3×10^{154}	1.6×10^{68}	5.2×10^{69}	1.6×10^{71}	5.2×10^{72}	1.6×10^{74}	5.2×10^{75}	1.6×10^{76}	8.8×10^{76}	1.4×10^{77}	1.9×10^{77}

Table shows number of hashes n(p) needed to achieve the given probability of success, assuming all hashes are equally likely. For comparison, 10^{-18} *to* 10^{-15} *is the uncorrectable bit error rate of a typical hard disk* [2]. *In theory, MD5 hashes or UUIDs, being 128 bits, should stay within that range until about 820 billion documents, even if its possible outputs are many more.*

It is easy to see that if the outputs of the function are distributed unevenly, then a collision could be found even faster. The notion of 'balance' of a hash function quantifies the resistance of the function to birthday attacks (exploiting uneven key distribution) and allows the vulnerability of popular hashes such as MD and SHA to be estimated (Bellare and Kohno, 2004 [3]).

The subexpression $\ln \frac{1}{1-p}$ in the equation for $n(p; H)$ is not computed accurately for small p when directly translated into common programming languages as `log(1/(1-p))` due to loss of significance. When `log1p` is available (as it is in ANSI C) for example, the equivalent expression `-log1p(-p)` should be used instead. If this is not done, the first column of the above table is computed as zero, and several items in the second column do not have even one correct significant digit.

Source code example

Here is a C++ program that can accurately generate most of the above table.

```
#include <math.h>
#include <stdlib.h>
#include <iostream>

/*
$ g++ -o birthday birthday.cc
$ ./birthday -15 128
8.24963e+11
$ ./birthday -6 32
92.6819
*/

int main(int argc, char ** argv) {
  if (argc != 3) {
    std::cerr << "Usage: " << argv[0] << " probability-exponent bits" << std::endl;
    return 1;
  }

  long probabilityExponent = strtol(argv[1], NULL, 10);
  double probability = pow(10, probabilityExponent);

  long bits = strtol(argv[2], NULL, 10);
  double outputs = pow(2, bits);

  std::cout << sqrt(2.0 * outputs * -log1p(-probability)) << std::endl;

  return 0;
}
```

Simple approximation

A good rule of thumb which can be used for mental calculation is the relation

$$p(n) \approx \frac{n^2}{2m}$$

which can also be written as

$$n \approx \sqrt{2m \times p(n)}.$$

This works well for probabilities less than or equal to 0.5.

This approximation scheme is especially easy to use for when working with exponents. For instance, suppose you are building 32-bit hashes ($m = 2^{32}$) and want the chance of a collision to be at most one in a million ($p \approx 2^{-20}$), how many documents could we have at the most?

$$n \approx \sqrt{2 \times 2^{32} \times 2^{-20}} = \sqrt{2^{1+32-20}} = \sqrt{2^{13}} = 2^{6.5} \approx 90.5$$

which is close to the correct answer of 93.

Digital signature susceptibility

Digital signatures can be susceptible to a birthday attack. A message m is typically signed by first computing $f(m)$, where f is a cryptographic hash function, and then using some secret key to sign $f(m)$. Suppose Mallory wants to trick Bob into signing a fraudulent contract. Mallory prepares a fair contract m and a fraudulent one m'. She then finds a number of positions where m can be changed without changing the meaning, such as inserting commas, empty lines, one versus two spaces after a sentence, replacing synonyms, etc. By combining these changes, she can create a huge number of variations on m which are all fair contracts.

In a similar manner, Mallory also creates a huge number of variations on the fraudulent contract m'. She then applies the hash function to all these variations until she finds a version of the fair contract and a version of the fraudulent contract which have the same hash value, $f(m) = f(m')$. She presents the fair version to Bob for signing. After Bob has signed, Mallory takes the signature and attaches it to the fraudulent contract. This signature then "proves" that Bob signed the fraudulent contract.

The probabilities differ slightly from the original birthday problem, as Mallory gains nothing by finding two fair or two fraudulent contracts with the same hash. Mallory's strategy is to generate pairs of one fair and one fraudulent contract. The birthday problem equations apply where n is the number of pairs. The number of hashes Mallory actually generates is $2n$.

To avoid this attack, the output length of the hash function used for a signature scheme can be chosen large enough so that the birthday attack becomes computationally infeasible, i.e. about twice as many bits as are needed to prevent an ordinary brute-force attack.

Pollard's rho algorithm for logarithms is an example for an algorithm using a birthday attack for the computation of discrete logarithms.

Notes

[1] See upper and lower bounds.
[2] http://arxiv.org/abs/cs/0701166
[3] http://citeseer.ist.psu.edu/bellare02hash.html

References

- Mihir Bellare, Tadayoshi Kohno: Hash Function Balance and Its Impact on Birthday Attacks. EUROCRYPT 2004: pp401–418
- *Applied Cryptography, 2nd ed.* by Bruce Schneier

External links

- "What is a digital signature and what is authentication?" (http://www.rsasecurity.com/rsalabs/node.asp?id=2182) from RSA Security's crypto FAQ.
- "Birthday Attack" (http://x5.net/faqs/crypto/q95.html) X5 Networks Crypto FAQs

Article Sources and Contributors

License