Direct Meet-in-the-Middle Attack

Charles Bouillaguet

v1.02 — November 19th 2024

CHANGELOG:

- Fixed the pseudo-code (Thanks to Philippe Gueorguievskii for pointing out the typo)

We provide you with a sequential C program that solves the following problem. Given two functions $f, g : \{0,1\}^n \to \{0,1\}^n$ and a predicate $\pi : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}$, find a "golden collision" (x, y) such that $f(x) = g(y)$ and $\pi(x, y) = 1$.

This can be done by brute force: testing all the 2^{2n} pairs (x, y). But there is also a relatively simple algorithm that does (in average) $3 \cdot 2^n$ "operations" and that uses 2^n "words" of memory (this assumes that the functions f and g are "somewhat random", or at least without observable structure). This is much better than naive brute-force. It works as follows:

```
Initialize a dictionary $\mathcal{D}$
for each $x \in \{0,1\}^n$ do
    Add the key-value pair $f(x) \mapsto x$ to the dictionary $\mathcal{D}$.
for each $y \in \{0,1\}^n$ do
    $L \leftarrow \{x \ : \ (g(y) \mapsto x) \in \mathcal{D}\}$          ▷ Retrieve all values associated with the key $g(y)$
    for each $x \in L$ do
        if $\pi(x, y)$ then
            return $(x, y)$
```

This procedure was invented in the context of cryptanalysis, where it has many applications, hence its name: the "meet-in-the-middle attack".

However, we stress that *no knowledge of cryptology is required* to do this project. Cryptographic techniques are used (by us) to create instances of the problem, but you can think of it purely as a (distributed) datastructure problem.

1 What you Have to Do

We provide you with

- A `mitm.c` sequential C program that takes as argument the description of (f, g, π). It searches a "golden collision" using the meet-in-the-middle attack. In other terms, it returns (x, y) such that $f(x) = g(y)$ and $\pi(x, y) = 1$.

- A *webservice* that provides you instances of the problem with the promise that a solution exists[1]:

  `https://ppar.tme-crypto.fr/<username>/<n>`

  Replace `<username>` by something like `firstname.lastname` and use an appropriate value of `<n>`. Example: `https://ppar.tme-crypto.fr/toto/42`.

Your goal is to find a golden collision for the largest possible input size (value of n), and to do so as fast as possible (but increasing n is more important). $n \geq 40$ begins to be seriously challenging. If you don't parallelize the sequential program we give you, you will be fairly limited by both space and time.

You may use MPI and OpenMP, and you can try to vectorize some bits of the code. It is expected that you end up with a parallel program capable of running on more than one compute node.

There is *one* technique that you must know about to build a distributed dictionary without a major headache: *sharding*. Google and ChatGPT are your friends.

You *can* try to use time-memory tradeoffs. But *do not* try to implement the Parallel collision search algorithm of Van Oorschott and Wiener (it's a recipe for failure: it seems simple, but making it work in practice is very, very hard).

Whatever you do, you **must**:

1. *Describe* what you have done.
2. *Measure* the performance of your implementation and *comment* it (what is the bottleneck?)

Your work must be submitted by Tuesday, January 7th before 23:59 (using Moodle). You must submit:

- The golden collisions you have found (along with n and the username so that we can check them).
- The source code used to find them.
- A `Makefile` that compiles your code (without errors nor warnings, even with `-Wall`).
- A ≈ 5-page report in `.PDF` format that explains what you have done. Please follow our guidelines about writing reports (on Moodle).
- If you run large computations in non-interactive mode (recommended), then you can also provide us with the output of your runs.

Some last notes:

- You should work in pairs (exceptions should be discussed with us). Submit one report with both names.
- You **MUST** respect the grid5000 usage policy. Do big computations at night/week-ends.
- If you believe that you have found an error in our programs (it does happen), or if you and your classmates are facing a common technical problem, don't hesitate to contact us.

[1] You have to trust us about this, at least for now. See you in the "cryptology 2" course next year for ways to also receive a *proof* that a solution exists...

2 Context

This section describes the origin of the problem and the cryptographic context; reading it is not required to do the project.

In cryptology, a *block cipher* is a family of pseudo-random permutations. It takes as input a k-bit *key* and an n-bit *plaintext block*, and produces an n-bit *ciphertext block*. This essentially *encrypts* n bits of data under the given k-bit key. Given the key, decryption is possible because the inverse permutation is available. More precisely, a block cipher is given by two functions:

$$E : \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$$
$$D : \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n$$

Such that

$$\forall K \in \{0,1\}^k,\ \forall x \in \{0,1\}^n \ : \ D(K, E(K,x)) = x.$$

(the idea is that E encrypts, while D decrypts).

Block ciphers are widely used to encrypt data, for instance to guarantee the confidentiality of "secure" internet connections. They belong to the realm of "symmetric cryptology", because the key used to encrypt the data is required to decrypt it. The encryption algorithm is considered secure if anyone who obtain encrypted data only observe pseudo-random garbage that has no meaning. In particular, nobody must be able to recover the encryption key by looking at the encrypted data (otherwise they would be able to decrypt the data using the normal decryption procedure...).

An attacker that would like to break the confidentiality (e.g. to "read" the encrypted data) could try to run an *attack* against the encryption algorithm. One of the simplest attacks is *exhaustive search*: trying all the possible keys one after the other. For instance, if an attacker possesses a *plaintext-ciphertext pair*, namely two n-bit blocks x and y such that $y = E(K, x)$, then they can run the following simple procedure: for each potential k-bit key K', check if $E(K', x)$ is equal to y; if so, K' is very likely to be the right key. This simple algorithm requires 2^k "operations". The size of the key (the number of bits k) must be chosen so that it is completely impractical: it must be beyond mankind's computational capabilities.

In 1975, the American government (IBM and the NSA) published the specification of an innovative block cipher called the DES (Data Encryption Standard). They claimed that it was secure and encouraged large American companies to use it instead of their own "homebrew" cryptographic algorithms (that were very often complete crap). At the time, research in cryptology was confined to military institutions. The publication of the complete description of a "modern" and secure encryption algorithm was akin to the discovery of a piece of alien technology for (some) researchers in computer science. This essentially kick-started public research in cryptology, at least to try to figure out whether the DES was actually secure.

The DES encrypted 64-bit blocks using 56-bit keys. The block size is fine, but the keys were quite short, and no justification was given for their size. The NSA would not tell *why* they had chosen 56-bit keys instead of, say, 128-bit keys. This was immediately criticized as suspicious. The main argument was that running the exhaustive search would be infeasible for ordinary people at the time, but could potentially be possible for the NSA because of its (secret) large computers and technological advance.

Fifty years later, the DES turned out to be a good and secure encryption algorithm... with keys that are way, *way* too short. It can no longer be used as-is nowadays because the "natural" increase in computational power has made exhaustive search practically feasible (with a machine that costs less than a small car in 2024).

After the publication of the DES, several researchers quickly proposed solutions to have longer keys. Two such ideas are the "double-DES" (E_2) and the "two-key triple-DES" (E_3):

$$E_2(K_1, K_2, x) := E(K_2, E(K_1, x)), \tag{1}$$
$$E_3(K_1, K_2, x) := E(K_2, D(K_1, E(K_2, x))). \tag{2}$$

However, it was quickly pointed out in 1977, that the double-DES was much less secure than expected. Even though it has 112-bit keys, it can be "broken" by a relatively simple procedure that requires only $2^{57.6}$ operations. This procedure became famously known as the "*meet-in-the-middle attack*". To run it, the attacker needs two plaintext-ciphertext pairs (P_0, C_0) and (P_1, C_1) with $C_i = E(K_2, E(K_1, P_i))$. The main observation is that we always have $D(K_2, C_0) = E(K_1, P_0)$. In other terms, encrypting P_0 using K_1 yields the "state in the middle", and so does decrypting C_0 using K_2.

The main idea of the meet-in-the-middle attack is to encrypt P_0 using all possible keys (for K_1), then check this against the decryption of C_0 using all possible keys (for K_2). Each match ("in the middle") suggests a potentially good pair of keys. Each pair of key must be tested against the other plaintext-ciphertext pairs for confirmation.

This is a special case of the problem given in the introduction, with

$$\begin{aligned} f(x) &= E(x, P_0) \\ g(y) &= D(y, C_0) \\ \pi(x, y) &= \left[E(y, E(x, P_1)) == C_1\right]. \end{aligned}$$