# NSUCRYPTO 2020: Collisions

Ioan Dragomir[1], Gabriel Tulba-Lecu[2], and Mircea-Costin Preoteasa[3]

[1] ioandr@gomir.pw – Technical University of Cluj-Napoca
[2] gabi_tulba_lecu@yahoo.com – Polytechnic Univeristy of Bucharest
[3] mircea_costin84@yahoo.com – Polytechnic Univeristy of Bucharest

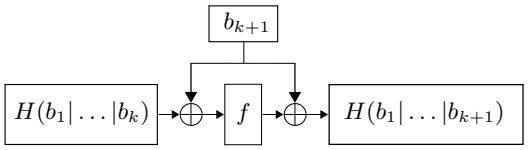
## Table of Contents



## 1 Problem summary

The following hash construction is proposed. It is based on an arbitrary $n$ bit to $n$ bit function $f$, operates on indefinitely many $n$ bit blocks $b_k$, and outputs an $n$ bit hash. The hash value starts as zero, and each added block updates it according to the following formula and diagram. $f$ is considered unknown.

$$H(b_1|\dots|b_k|b_{k+1}) = b_{k+1} \oplus f(b_{k+1} \oplus H(b_1|\dots|b_k))$$




***Question 1:*** Propose an algorithm which finds a collision for $H$.

***Question 2:*** Propose an algorithm which, given a message $m$ and $H(m)$, finds a second preimage $m' \neq m,\ H(m') = H(m)$.

***Question 3:*** Given $n = 32$ and the message "`A random matrix is likely decent`", $m =$ (`0x412072616e646f6d206d617472697820697320c696b656c7920646563656e74`), find another message $m'$ with the same hash. We are also given the possibility to hash arbitrary messages and a precomputed list of $(x, f(x))$ pairs for $x \in \{0, \dots, 511\}$.

## 2 Solution summary

We will focus on solving question 2, as answers for questions 1 and 3 can be obtained very easily given a working construction for a second preimage attack. For this, we propose two solutions, which we discovered separately, but we believe to be equivalent. The first is a naive exploration of how to add two blocks so that together they don't change the hash, and the second manages the same, but is based on an actual mathematical proof, and is much more computationally efficient. By repeatedly applying either algorithm, we can extend the message by an even number of blocks without changing its hash.

## 3 Exploratory solution

As discussed above, we will look for solutions which append two blocks, $b_1, b_2$, the first of which inevitably changes the hash, so the second must bring it back to $H_0$.

$$H_1 = b_1 \oplus f(b_1 \oplus H_0)$$
$$H_2 = b_2 \oplus f(b_2 \oplus H_1) \stackrel{!}{=} H_0$$

We notice $b_1 \oplus H_0$ must be one of the known $x$ values, and so does $b_2 \oplus H_1$. Let $a_1 = b_1 \oplus H_0$. We can now iterate through values for $a_1$ for which we know $f(a_1)$, and calculate the corresponding $b_1 = a_1 \oplus H_0$ afterwards:

```
for a1 in known_x:
        b1 = a1 ^ h0
        h1 = b1 ^ known_f[b1 ^ h0]
```

For each iterated $a_1$, we can do a similar loop through $a_2 = b_2 \oplus H_1$, and look for the values where $H_2 = H_0$:

```
        for a2 in known_x:
                b2 = a2 ^ h1
                h2 = b2 ^ known_f[b2 ^ h1]
                if h2 == h0:
                        print("Solution:", b1, b2)
```

As it stands, this algorithm is quadratic in the number of known $f$ points. We can optimise the second loop by manipulating the second condition:

$$\begin{aligned} H_2 &= H_0 \\ b_2 \oplus f(b_2 \oplus H_1) &= H_0 \\ H_1 \oplus b_2 \oplus f(b_2 \oplus H_1) &= H_1 \oplus H_0 \\ q \oplus f(q) &= H_1 \oplus H_0, \quad q = b_2 \oplus H_1 \end{aligned}$$

We can build a lookup table to quickly compute such $x \oplus f(x) = y$ equations:

```
xfx = { x^known_f[x] : x for x in known_x }
```

And simplify the inner loop from a $O(m)$ loop to a $O(\log m)$ lookup:

```
for a1 in known_x:
        b1 = a1 ^ h0
        h1 = b1 ^ known_f[b1 ^ h0]
        if h0 ^ h1 not in xfx: continue
        q = xfx[h0 ^ h1]
        b2 = q ^ h1
        print("Solution:", b1, b2)
```

Experimentally we notice there are always $m$ solutions, suggesting that each $(x, f(x))$ pair generates one.

## 4 Rigorous solution

While playing around with the previous algorithm, Ioan had missed multiple possible mathematical gadgets, whereas Gabi promptly used them to get to a much more elegant algorithm. We define $g(x) = x \oplus f(x)$ and write the hash step function in terms of $g$:

$$\begin{aligned}
H_{k+1} &= b_{k+1} \oplus f(b_{k+1} \oplus H_k) \\
H_{k+1} \oplus H_k &= b_{k+1} \oplus H_k \oplus f(b_{k+1} \oplus H_k) \\
H_{k+1} &= H_k \oplus g(b_{k+1} \oplus H_k) \\
H_{k+2} &= H_{k+1} \oplus g(b_{k+2} \oplus H_{k+1}) \\
H_{k+2} &= H_k \oplus g(b_{k+1} \oplus H_k) \oplus g(b_{k+2} \oplus H_{k+1})
\end{aligned}$$

If we set the two $g$ terms to be identical, i.e. $b_{k+1} \oplus H_k = b_{k+2} \oplus H_{k+1}$, we get $H_{k+2} = H_k$, and thus a collision.

$$\begin{aligned}
b_{k+1} \oplus H_k &= b_{k+2} \oplus H_{k+1} \\
b_{k+1} \oplus H_k &= b_{k+2} \oplus H_k \oplus g(b_{k+1} \oplus H_k) \\
b_{k+2} &= b_{k+1} \oplus g(b_{k+1} \oplus H_k) \\
b_{k+2} &= b_{k+1} \oplus b_{k+1} \oplus H_k \oplus f(b_{k+1} \oplus H_k) \\
b_{k+2} &= H_k \oplus f(b_{k+1} \oplus H_k)
\end{aligned}$$

Thus, given a message with hash $H_0$, we can generate a collision for each known $(x, f(x))$ pair by appending the blocks:

$$\begin{aligned}
b_{k+1} &= H_0 \oplus x \\
b_{k+2} &= H_0 \oplus f(x)
\end{aligned}$$

## 5 Efficiency comparison

Considering $n$ is the number of bits, $p$ is the number of known $(x, f(x))$ pairs, ($p = 10n$ for questions 1, 2; $p = 2n$ for question 3) which are both very small values in practice.

The first solution has a memory complexity of $O(p)$, an upfront time complexity of $O(p)$ and an additional $O(\log n)$ lookup for each of the $p$ collisions possible by its construction.

The second solution has $O(1)$ time and $O(1)$ memory complexity for a single collision. For all $m$ collisions, it's $O(p)$ linear time but still constant memory.

Therefore the second solution is a theoretical winner, though in practice the relatively small values of $n$ and $p$ make their performance barely distinguishable.

## 6 Question 1

Question 1 does not suppose we already have a message–hash pair, thus we must create one only using the known $(x, f(x))$ pairs. For some known $x$ interpreted as a block $m = (x)$, we have $H(m) = x \oplus f(x)$, which is computable. We then apply the discovered construction to obtain a 3-block message $m' = m \mid\mid H(x) \oplus x \mid\mid H(x) \oplus f(x)$ with the same hash as $m$.

## 7 Question 3

We now apply the construction on:
$n = 32$
$m =$ `0x412072616e646f6d206d617472697820697320c696b656c7920646563656e74`
$H(m) =$ `0xd64adbdf7458e158801ef33370d0a7c524065545447517d39ccbb0d8d9015bdc`
and using any of the 512 given $(x, f(x))$ pairs, for instance
$f(123) =$ `0x58720e441a8a4fb6c0cd564227b1caf9df3d854c7a097cbf3403eb81d45dca6`
we can determine $m' = m \mid\mid H(m) \oplus 123 \mid\mid H(m) \oplus f(123)$ which is, in hexadecimal:

```
412072616e646f6d206d617472697820697320c696b656c7920646563656e74
d64adbdf7458e158801ef33370d0a7c524065545447517d39ccbb0d8d9015ba7
d3cdfb3b35f045a3ec12265752abbb6ab9f58d1183d580186f8b8e60c444877a
```

We can also verify that the calculated message: `h412072...44877a` results in the same hash on the hash evaluator form.

## 8 Further research: printable second preimages

We dearly hoped we would be able to find a truly excellent construction which would allow us to extend a message using only fully printable blocks (i.e. in ASCII, blocks where each byte is between 32 and 126 inclusive). The expression

$$H_{k+j} = H_k \oplus g(b_{k+1} \oplus H_k) \oplus g(b_{k+2} \oplus H_{k+1}) \oplus \cdots \oplus g(b_{k+j} \oplus H_{k+j-1})$$

seems promising, as it expresses an updated hash just as a linear combination, but complexity quickly arises when we consider each of the $g$ terms depends on everything to its left. It feels like we're quite close, but at the same time, there's no solution in sight. Very unfortunate! It would have made for an extraordinary demonstration.