On the Meet-in-the-Middle Attack

Ivica Nikolić
(joint with Yu Sasaki)

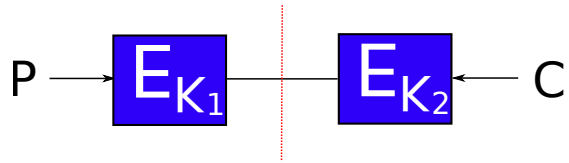
NTU, Singapore

Meet-in-the-middle attack

- Introduced by Diffie-Hellman
- Recover the 2 keys of 2DES by finding a collision in the middle of the cipher

$$E_{K_2} \circ E_{K_1}(P) = C$$

Meet-in-the-middle attack

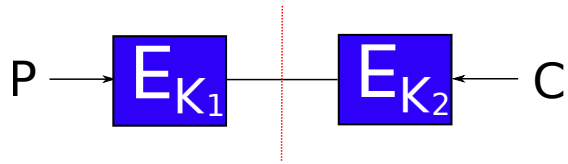


Problem: $E_{K_1}(P) = E_{K_2}^{-1}(C)$

Given two pairs $(P_1, C_1), (P_2, C_2)$:

- For all keys K_1 save in a hash table H the values $(E_{K_1}(P_1), E_{K_1}(P_2), K_1)$
- For each key K_2 check if $(E_{K_2}^{-1}(C_1), E_{K_2}^{-1}(C_2))$ is in H
- Match corresponds to the secret key $K_1||K_2$

Meet-in-the-middle attack



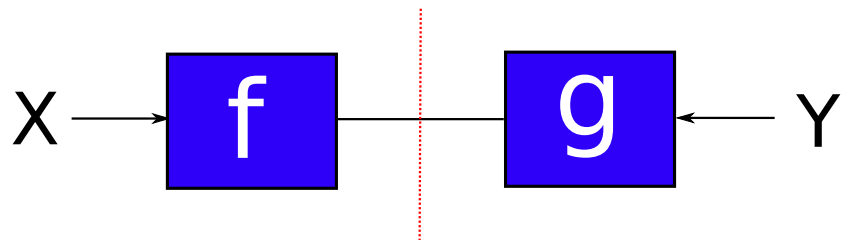
Problem: $E_{K_1}(P) = E_{K_2}^{-1}(C)$

To recover the $2n$-bit key $K_1||K_2$:

- Time: 2^n to create H and 2^n to find a match
- Memory: 2^n to save H

MITM = Collision

- Today, MITM attack is synonym for collision search between two functions.
- Example, MITM attacks on AES have nothing to do with "meeting in the middle".
- So, instead of MITM we can talk about collisions.



Collisions: Types

Usually, there are two types of collisions between f and g

1. f, g **have range larger than domain.**
   Recover the unique 2 keys of 2DES given 2 pairs (P, C).
2. **Range not larger than domain.**
   Find some 2 keys of 2DES given 1 pair (P, C).

Our target: Unbalanced Collisions

Deal only with the case 2.
To simplify, focus only on collision search between two functions f, g:

$$f : \{0,1\}^n \to \{0,1\}^n$$
$$g : \{0,1\}^n \to \{0,1\}^n$$

Unbalanced Collisions – g is R times more "expensive" than f

State-of-the-art

When $R = 1$ (f, g have the same cost) then

- use Floyd's cycle finding algorithm
- it requires time $T = 2^{\frac{n}{2}}$
- it requires **negligible** memory

State-of-the-art

When $R = 2^{\frac{n}{100}}$, then

- use DH MITM
- Store $2^{\frac{n}{2}-\frac{n}{200}}$ images of g
  (in time $2^{\frac{n}{2}-\frac{n}{200}} \times 2^{\frac{n}{100}} = 2^{\frac{n}{2}+\frac{n}{200}}$)
- Produce around $2^{\frac{n}{2}+\frac{n}{200}}$ images of f and check for collision
- Success because $2^{\frac{n}{2}-\frac{n}{200}} \times 2^{\frac{n}{2}+\frac{n}{200}} = 2^n$

Time: $2^{\frac{n}{2}+\frac{n}{200}}$
Memory: $2^{\frac{n}{2}-\frac{n}{200}}$

State-of-the-art

Table: Complexities

R	Time	Memory
1	$2^{\frac{n}{2}}$	negl
$2^{\frac{n}{100}}$	$2^{\frac{n}{2}+\frac{n}{200}}$	$2^{\frac{n}{2}-\frac{n}{200}}$

Tradeoff

In general, when $R > 1$,

$$T = 2^{\frac{n}{2}} \cdot \sqrt{R}$$
$$M = 2^{\frac{n}{2}} / \sqrt{R}$$

Tradeoff:

$$TM = 2^n$$

where $T \geq \sqrt{R} \cdot 2^{\frac{n}{2}}$.

Observation

- When $R = 1$ we use Floyd's algorithm $\Longrightarrow$ negl. memory
- When $R > 1$ we use standard MITM $\Longrightarrow$ huge memory

Find the missing link (algorithm):
The smaller R, the smaller memory requirement

General

New Algorithm

Conclusion

Ideas

New algorithm combines 2 ideas:

1. Unbalanced interleaving
2. van Oorschot-Wiener parallel collision search

Unbalanced interleaving

Balanced interleaving

Floyd's algorithm used for collision search of 2 balanced functions selects the used function with equal probability. I.e. it finds a collision for $F(x)$ defined as

$$F(x) = \begin{cases} f(x) & \text{if } \sigma(x) = 0 \\ g(x) & \text{if } \sigma(x) = 1 \end{cases}$$

$\sigma(x)$ outputs 0 or 1, with equal probability

Collisions for $F(x)$ is collision between f, g with probability $\frac{1}{2}$

$\implies$ repeat the search 2 times

Unbalanced interleaving

Unbalanced interleaving

Define $F(x)$ as

$$F(x) = \begin{cases} f(x) & \text{if } \sigma(x) = 0 \\ g(x) & \text{if } \sigma(x) = 1 \end{cases}$$

$\sigma(x)$ outputs 0 around R times more often than 1

Collisions for $F(x)$ is collision between f, g with probability $\frac{1}{R}$

$\implies$ repeat the search R times

van Oorschot-Wiener Parallel Collision Search

Multiple collisions for $f(x)$

- Floyd's algorithm with time $2^{\frac{n}{2}}$ is optimal to find one collision
- However, sometimes we need multiple collisions
- If s collisions are required, then Floyd needs $s \cdot 2^{\frac{n}{2}}$ time
- It needs negligible memory

van Oorschot-Wiener Parallel Collision Search

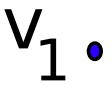
van Oorschot-Wiener algorithm can be used to find multiple collisions faster:

- Useful when s is larger
- It requires memory

van Oorschot-Wiener Algorithm: Hash Table

First, construct a hash table:

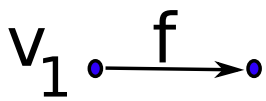
- Take a random point v_1 and produce a chain of values $v_i = f(v_{i-1}), i = 2, \ldots, 2^{\frac{n-m}{2}}$
- Store $(v_{2^{\frac{n-m}{2}}}, v_1)$ in hash table H
- Repeat for 2^m different points



van Oorschot-Wiener Algorithm: Hash Table

First, construct a hash table:

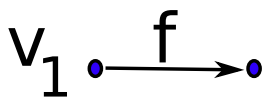
- Take a random point v_1 and produce a chain of values $v_i = f(v_{i-1}), i = 2, \ldots, 2^{\frac{n-m}{2}}$
- Store $(v_{2^{\frac{n-m}{2}}}, v_1)$ in hash table H
- Repeat for 2^m different points



van Oorschot-Wiener Algorithm: Hash Table

First, construct a hash table:

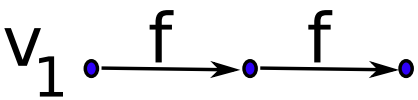
- Take a random point v_1 and produce a chain of values $v_i = f(v_{i-1}), i = 2, \ldots, 2^{\frac{n-m}{2}}$
- Store $(v_{2^{\frac{n-m}{2}}}, v_1)$ in hash table H
- Repeat for 2^m different points



van Oorschot-Wiener Algorithm: Hash Table

First, construct a hash table:

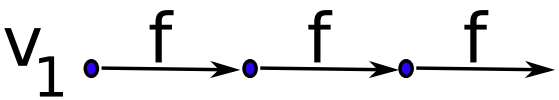
- Take a random point v_1 and produce a chain of values $v_i = f(v_{i-1}), i = 2, \ldots, 2^{\frac{n-m}{2}}$
- Store $(v_{2^{\frac{n-m}{2}}}, v_1)$ in hash table H
- Repeat for 2^m different points



van Oorschot-Wiener Algorithm: Hash Table

First, construct a hash table:

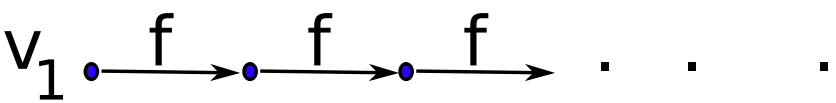
- Take a random point v_1 and produce a chain of values $v_i = f(v_{i-1}), i = 2, \ldots, 2^{\frac{n-m}{2}}$
- Store $(v_{2^{\frac{n-m}{2}}}, v_1)$ in hash table H
- Repeat for 2^m different points



van Oorschot-Wiener Algorithm: Hash Table

First, construct a hash table:

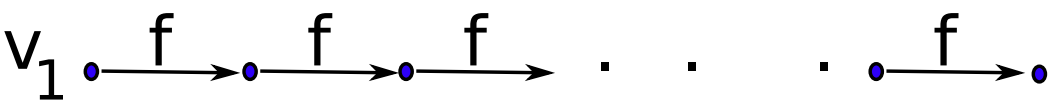
- Take a random point v_1 and produce a chain of values $v_i = f(v_{i-1}), i = 2, \ldots, 2^{\frac{n-m}{2}}$
- Store $(v_{2^{\frac{n-m}{2}}}, v_1)$ in hash table H
- Repeat for 2^m different points



van Oorschot-Wiener Algorithm: Hash Table

First, construct a hash table:

- Take a random point v_1 and produce a chain of values $v_i = f(v_{i-1}), i = 2, \ldots, 2^{\frac{n-m}{2}}$
- Store $(v_{2^{\frac{n-m}{2}}}, v_1)$ in hash table H
- Repeat for 2^m different points



van Oorschot-Wiener Algorithm: Hash Table

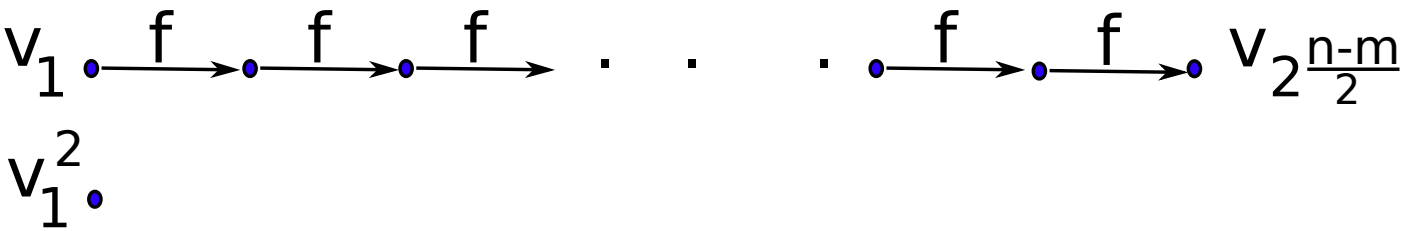
First, construct a hash table:

- Take a random point v_1 and produce a chain of values $v_i = f(v_{i-1}), i = 2, \ldots, 2^{\frac{n-m}{2}}$
- Store $(v_{2^{\frac{n-m}{2}}}, v_1)$ in hash table H
- Repeat for 2^m different points

van Oorschot-Wiener Algorithm: Hash Table

First, construct a hash table:

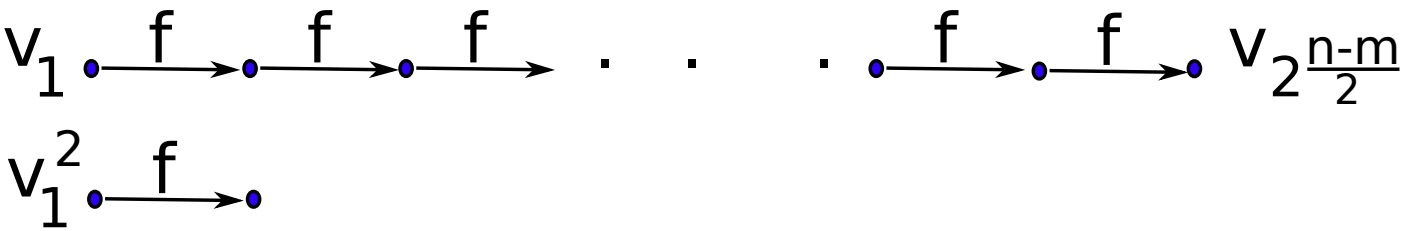
- Take a random point v_1 and produce a chain of values $v_i = f(v_{i-1}), i = 2, \ldots, 2^{\frac{n-m}{2}}$
- Store $(v_{2^{\frac{n-m}{2}}}, v_1)$ in hash table H
- Repeat for 2^m different points



van Oorschot-Wiener Algorithm: Hash Table

First, construct a hash table:

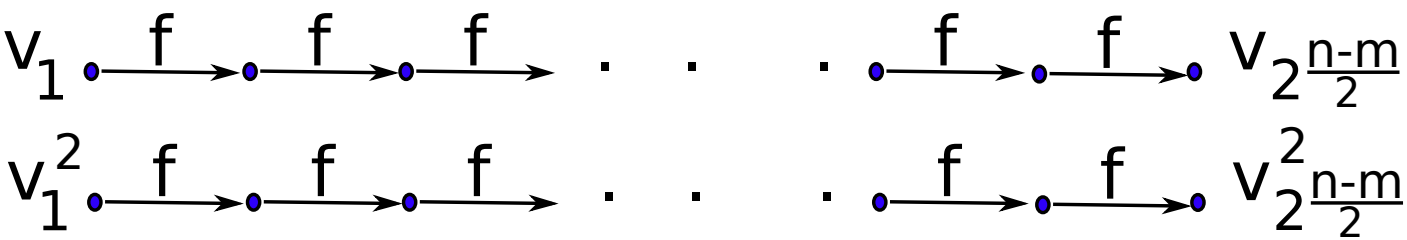
- Take a random point v_1 and produce a chain of values $v_i = f(v_{i-1}), i = 2, \ldots, 2^{\frac{n-m}{2}}$
- Store $(v_{2^{\frac{n-m}{2}}}, v_1)$ in hash table H
- Repeat for 2^m different points



van Oorschot-Wiener Algorithm: Hash Table

First, construct a hash table:

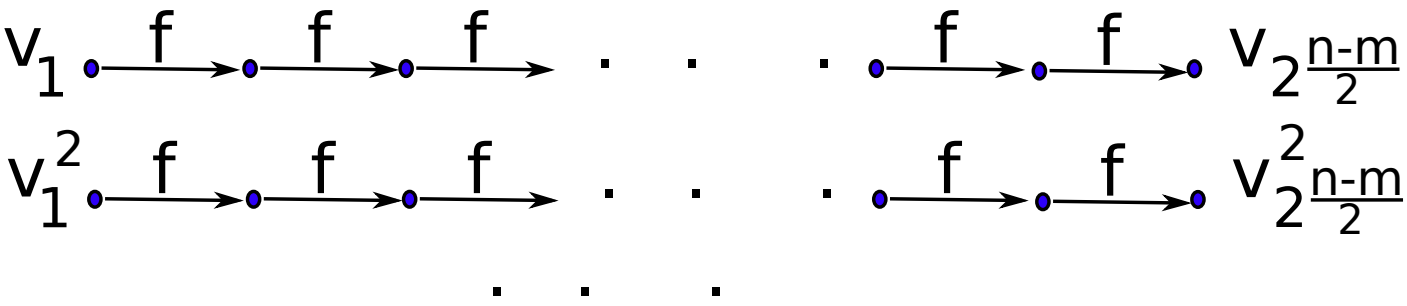
- Take a random point v_1 and produce a chain of values $v_i = f(v_{i-1}), i = 2, \ldots, 2^{\frac{n-m}{2}}$
- Store $(v_{2^{\frac{n-m}{2}}}, v_1)$ in hash table H
- Repeat for 2^m different points



van Oorschot-Wiener Algorithm: Hash Table

First, construct a hash table:

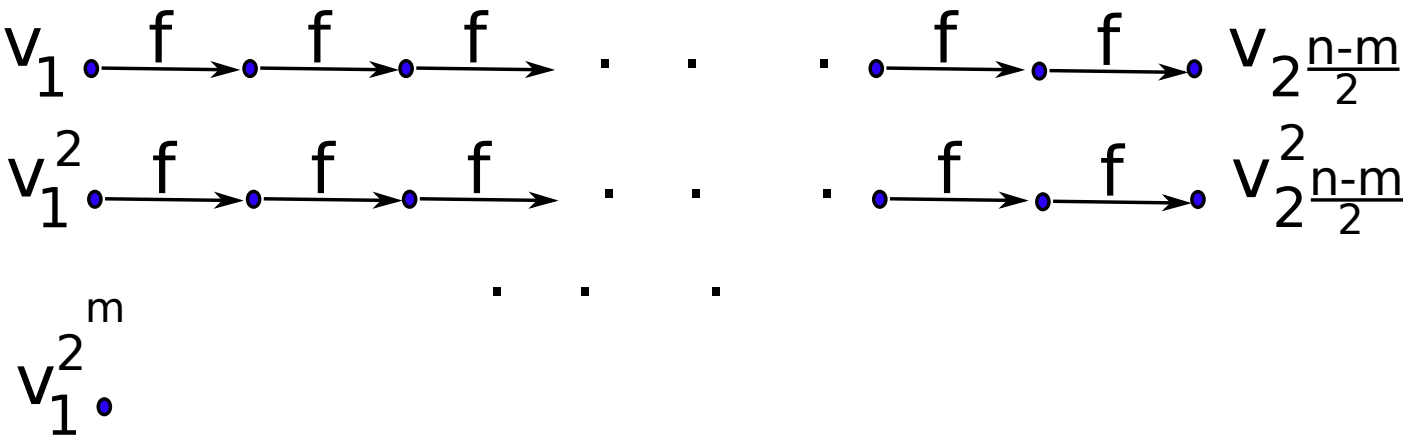
- Take a random point v_1 and produce a chain of values $v_i = f(v_{i-1}), i = 2, \ldots, 2^{\frac{n-m}{2}}$
- Store $(v_{2^{\frac{n-m}{2}}}, v_1)$ in hash table H
- Repeat for 2^m different points



van Oorschot-Wiener Algorithm: Hash Table

First, construct a hash table:

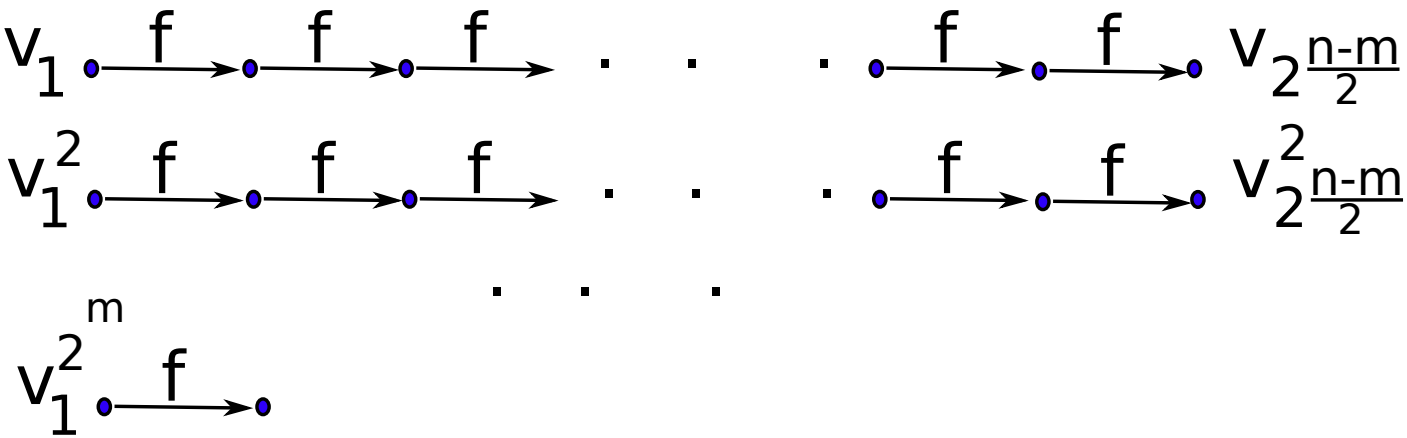
- Take a random point v_1 and produce a chain of values $v_i = f(v_{i-1}), i = 2, \ldots, 2^{\frac{n-m}{2}}$
- Store $(v_{2^{\frac{n-m}{2}}}, v_1)$ in hash table H
- Repeat for 2^m different points



van Oorschot-Wiener Algorithm: Hash Table

First, construct a hash table:

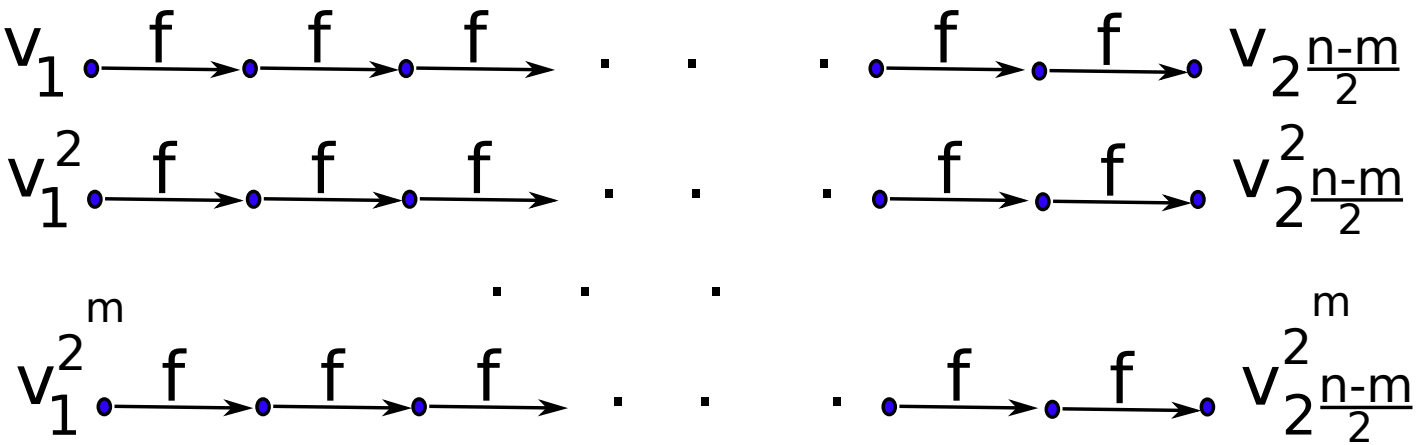
- Take a random point v_1 and produce a chain of values $v_i = f(v_{i-1}), i = 2, \ldots, 2^{\frac{n-m}{2}}$
- Store $(v_{2^{\frac{n-m}{2}}}, v_1)$ in hash table H
- Repeat for 2^m different points



van Oorschot-Wiener Algorithm: Hash Table

First, construct a hash table:

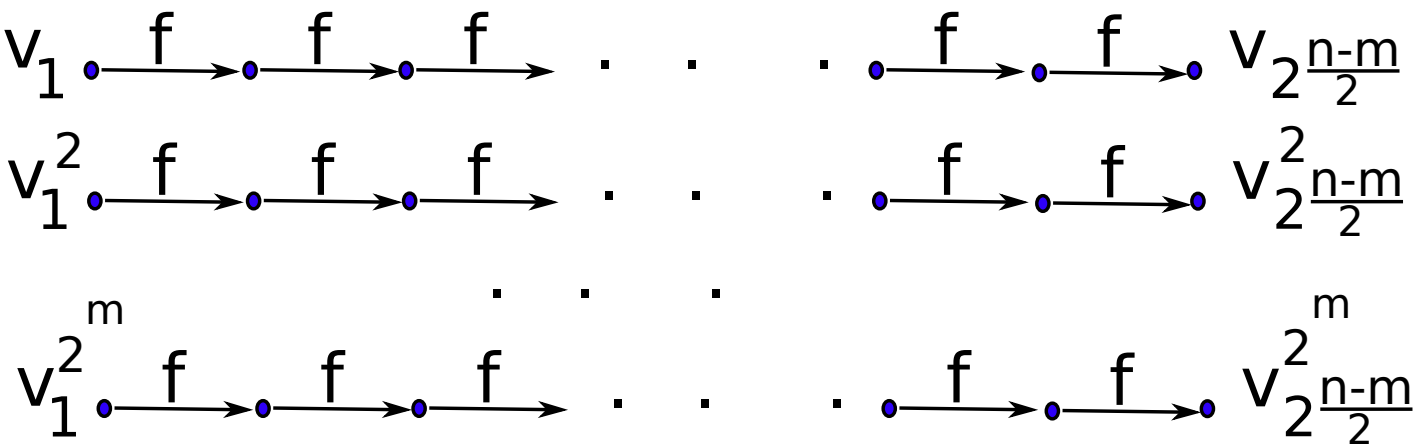
- Take a random point v_1 and produce a chain of values $v_i = f(v_{i-1}), i = 2, \ldots, 2^{\frac{n-m}{2}}$
- Store $(v_{2^{\frac{n-m}{2}}}, v_1)$ in hash table H
- Repeat for 2^m different points



van Oorschot-Wiener Algorithm: Hash Table

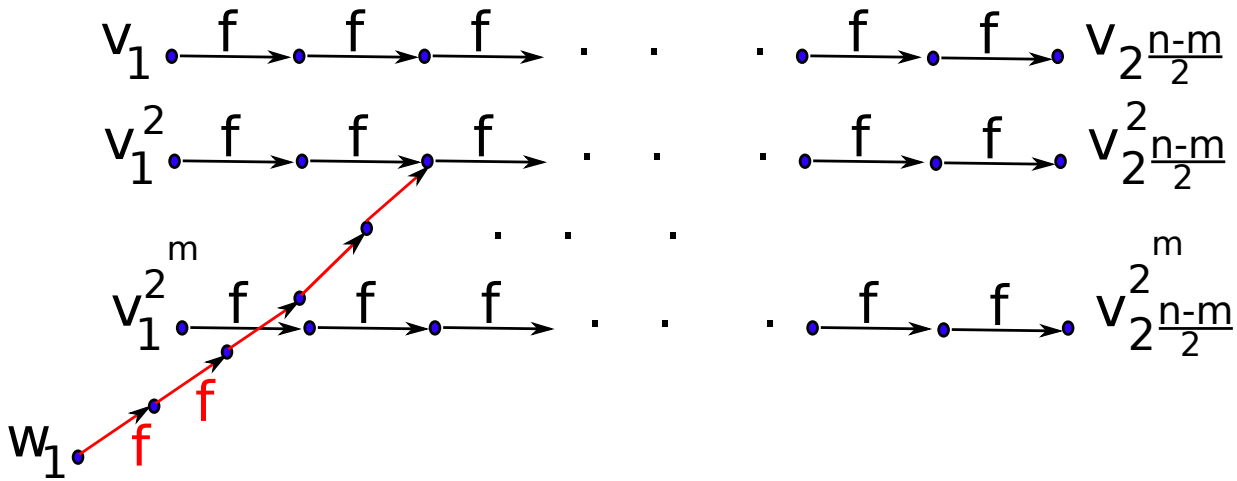
Construction of H:

- Time: $2^m \times 2^{\frac{n-m}{2}} = 2^{\frac{n+m}{2}}$ calls to $f(x)$
- Memory: 2^m



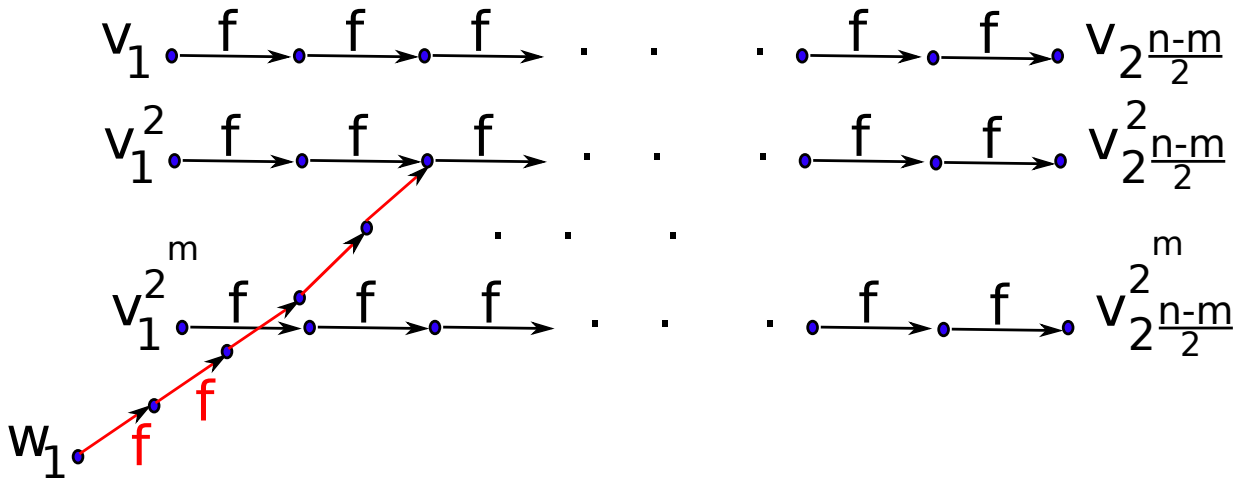
van Oorschot-Wiener Algorithm: Collision Search

1. Pick a random value w_1
2. Produce $w_i = f(w_{i-1})$
3. Check if w_i is in H. If not go to 2
4. By backtracking find the colliding values



van Oorschot-Wiener Algorithm: Collision Search

- During construction of H passed $2^{\frac{n+m}{2}}$ values
- If chain of w_i's is of length around $2^n/2^{\frac{n+m}{2}} = 2^{\frac{n-m}{2}}$ a collision will occur
- Time complexity of one collision: $2^{\frac{n-m}{2}}$



van Oorschot-Wiener Algorithm: Summary

- Initial cost for H: time $2^{\frac{n+m}{2}}$, memory 2^m
- Subsequent s collisions cost: $s \cdot 2^{\frac{n-m}{2}}$

When $2^{\frac{n+m}{2}} + s \cdot 2^{\frac{n-m}{2}} < s \cdot 2^{\frac{n}{2}}$
then van Oorschot-Wiener outperforms Floyd

Exampe: $s = 2^{\frac{n}{3}}$ collisions and $2^{\frac{n}{3}}$ memory

- van Oorschot-Wiener : $2^{\frac{2n}{3}}$
- Floyd: $2^{\frac{5n}{6}}$

New Algorithm for Unbalanced Collision Search

1. Define $F(x)$ as

$$F(x) = \begin{cases} f(x) & \text{if } \sigma(x) = 0 \\ g(x) & \text{if } \sigma(x) = 1 \end{cases}$$

   $\sigma(x)$ outputs 0 around R times more often than 1
2. Construct hash table H for $F(x)$ with $M = 2^m$ entries
3. Find collision for $F(x)$. If not a collision for f, g repeat step 3

After repeating 3. around R times, collision for f, g will appear

New Algorithm: Complexity

1. Define $F(x)$ as

$$F(x) = \begin{cases} f(x) & \text{if } \sigma(x) = 0 \\ g(x) & \text{if } \sigma(x) = 1 \end{cases}$$

   $\sigma(x)$ outputs 0 around R times more often than 1
2. Construct hash table H for $F(x)$ with $M = 2^m$ entries
3. Find collision for $F(x)$. If not a collision for f, g repeat step 3 (repeated R times)

Memory: $M = 2^m$

Time:

- Step 2:
  Calls to $F(x)$:$2^{\frac{n+m}{2}}$
  Calls to f: $2^{\frac{n+m}{2}}$
  Calls to g: $2^{\frac{n+m}{2}}/R$, but cost $2^{\frac{n+m}{2}}$
- Step 3:
  Calls to $F(x)$:$R \cdot 2^{\frac{n-m}{2}}$
  Calls to f: $R \cdot 2^{\frac{n-m}{2}}$
  Calls to g: $2^{\frac{n-m}{2}}$, but cost $R \cdot 2^{\frac{n-m}{2}}$

Total: $T = 2^{\frac{n+m}{2}} + R \cdot 2^{\frac{n-m}{2}}$

New Algorithm: Complexity

$$M = 2^m$$
$$T = 2^{\frac{n+m}{2}} + R \cdot 2^{\frac{n-m}{2}}$$

When $2^{\frac{n+m}{2}} \leq R \cdot 2^{\frac{n-m}{2}}$, i.e. when $M \leq R$, then $T \approx R \cdot 2^{\frac{n-m}{2}}$, thus

$$T^2 M = R^2 \cdot 2^{(n-m)+m} = R^2 \cdot 2^n$$

We end up with new tradeoff ($N = 2^n$):

$$T^2 M = R^2 N,$$

where $M \leq R$.

New Algorithm: Comparison to Standard Algorithm

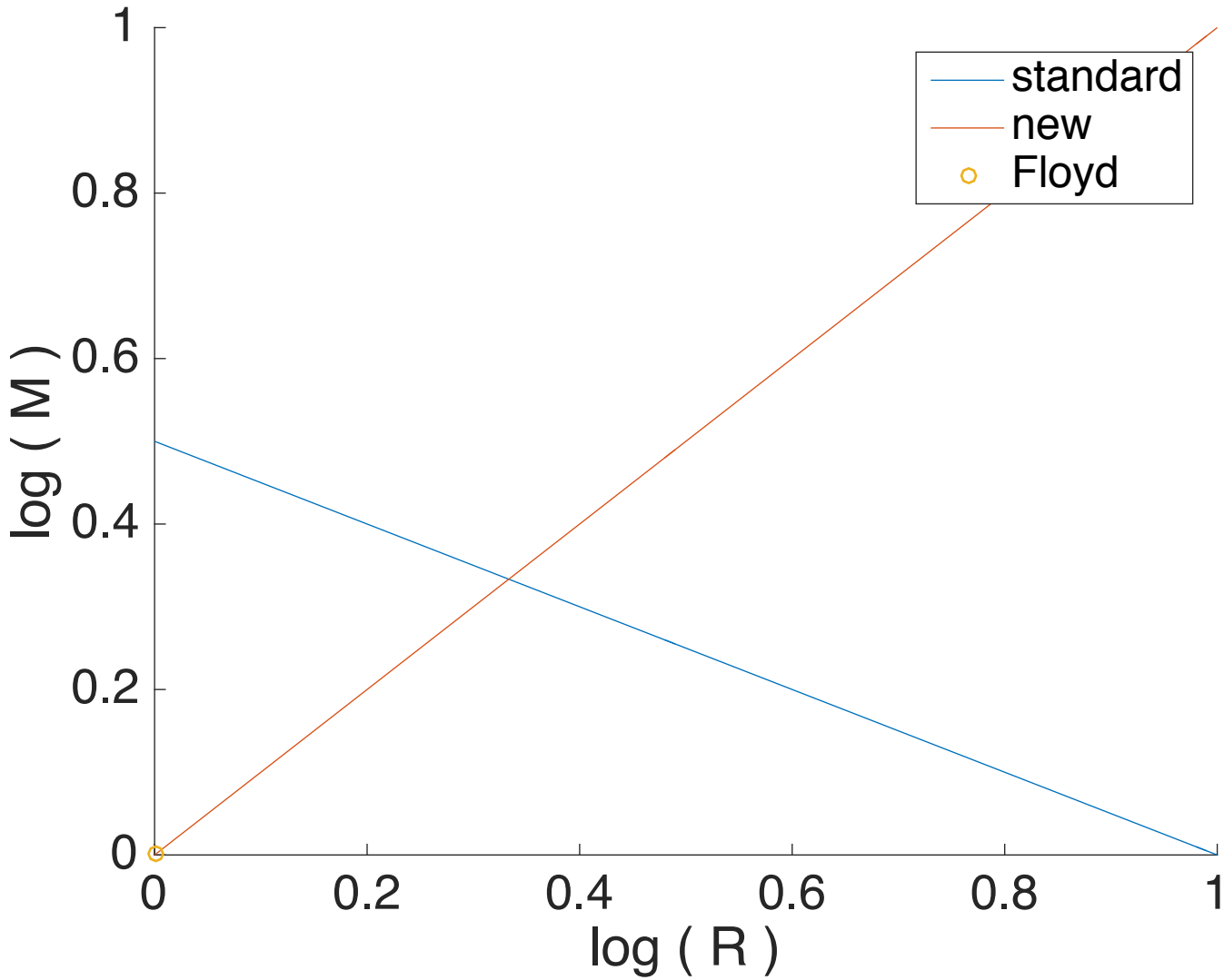
New $T^2M = R^2N$ against the standard $TM = N$

- **Better time** when $M < \frac{N}{R^2}$ (and $M \leq R$)
- **Better memory** when $T > R^2$ (and $T > \sqrt{RN}$)

If standard algorithm uses minimal time complexity $T < N^{\frac{2}{3}}$, then with new algorithm memory can be reduced from $\frac{N}{T}$ (which is $\geq N^{\frac{1}{3}}$) to $\frac{T^2}{N}$ (which is $< N^{\frac{1}{3}}$).

New Algorithm: The Missing Link

Smaller the ratio R, less memory is required by the new algorithm.



General

New AIgorithm

Conclusion

Conclusions

- Consider using the new algorithm when dealing with unbalanced MITM problems
- Rule of thumb: if $R \leq 2^{\frac{n}{3}}$ then most likely the memory complexity of your attack can be reduced with the new algorithm (without increasing the time)
- There are numerous applications of the new algorithm: check our paper on eprint