Recent Applications of Hellman's Time-Memory Tradeoff

Yu Sasaki

NTT Secure Platform Laboratories
Nanyang Technological University, Singapore

1/October/2015 @ ASK 2015, Singapore

Topics

This talk focuses on a cryptanalytic tool:

Hellman's time-memory tradeoff

Motivation

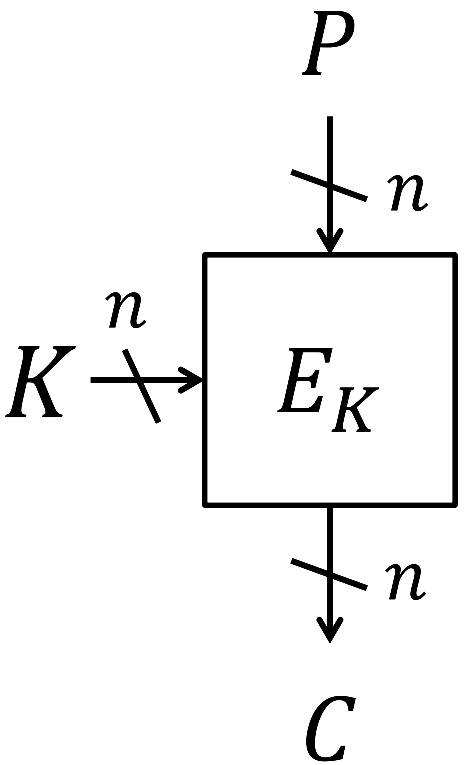
- Low memory attack is a recent trend
- Recently, I have found two applications:
  1. NMAC/HMAC key recovery (CRYPTO'14)
  2. Generalized birthday problem (Asiacrypt'15)

Hellman's Time-Memory Tradeoff

Introduction of Hellman's Tradeoff

"A Cryptanalytic Time-Memory Trade-Off."

Martin E. Hellman, 1980.



Key Recovery against Block Cipher

[Offline]

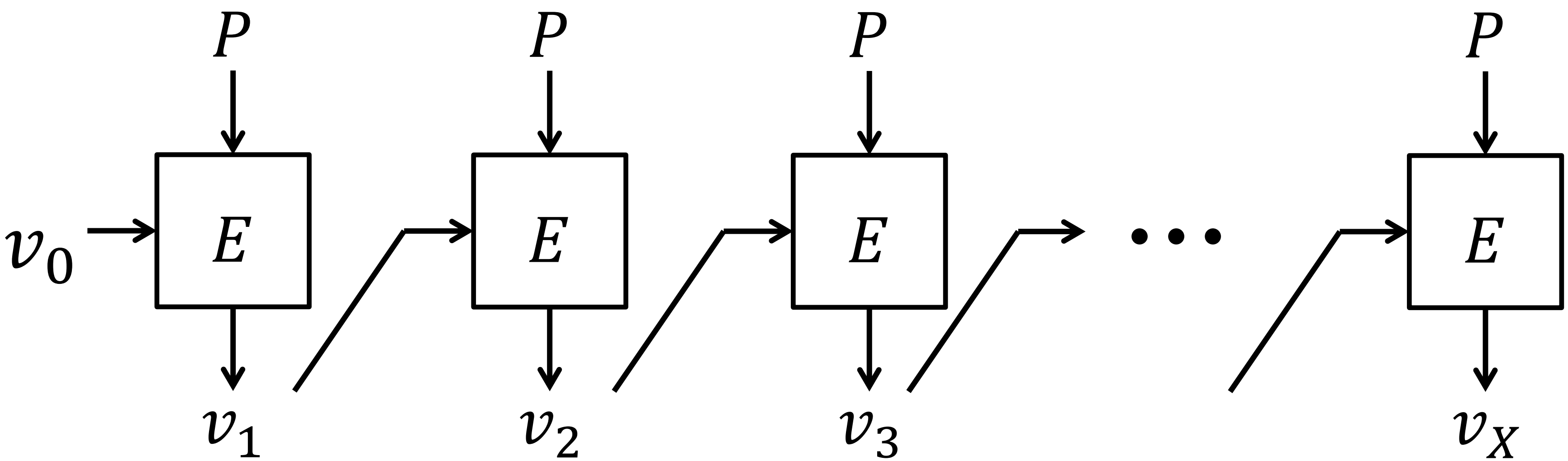
- 2^n precomp, $< 2^n$ memory

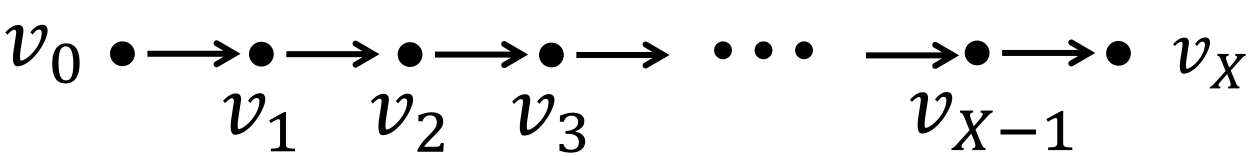
[Online]

- Any key can be recovered with complexity less than 2^n

Chains with Key Values

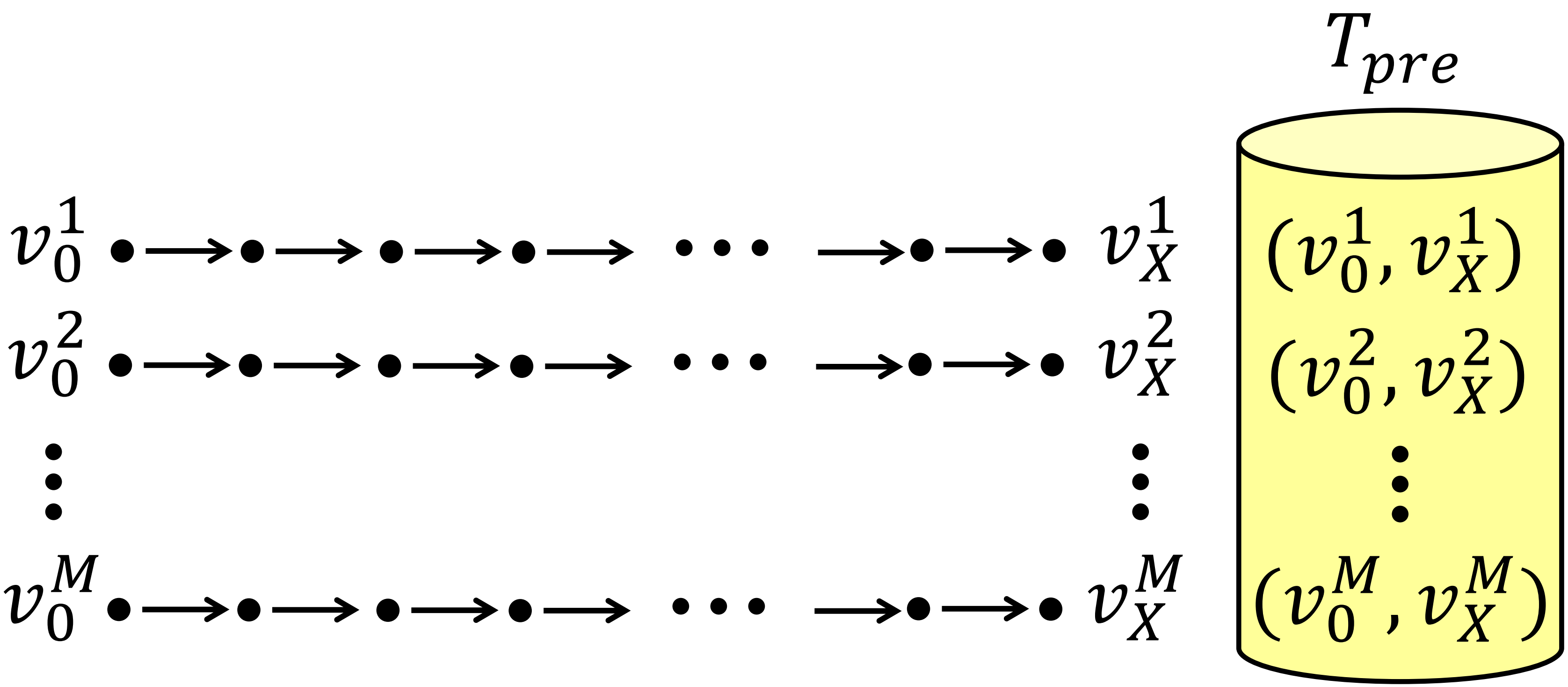
- Randomly choose a plaintext P
- Randomly choose starting key value v_0.
- Make chains of key values for X blocks.





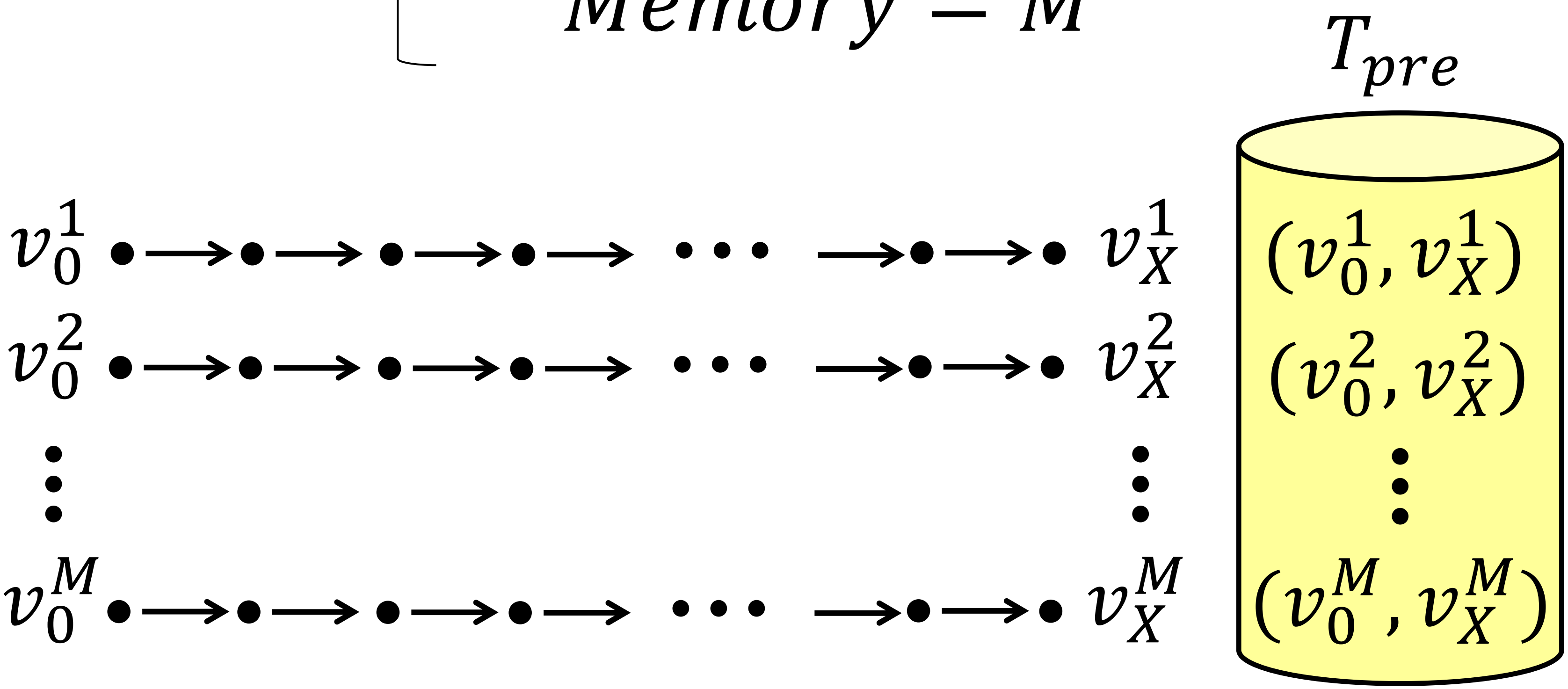
Many Chains with Saving Memory

- M chains of length X s.t. $M \times X = 2^n$
- Only start and end points are stored in T_{pre}



- (Ideally) all key values appear in chains.

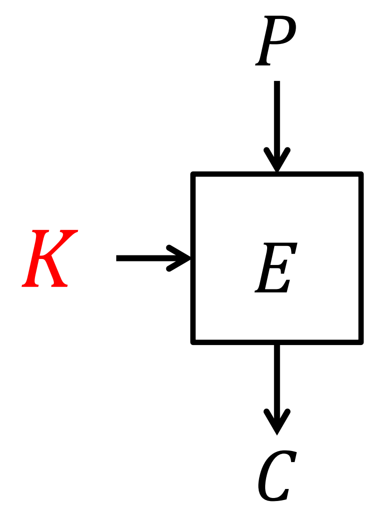
$$\begin{cases} Time = MX = 2^n \\ Memory = M \end{cases}$$

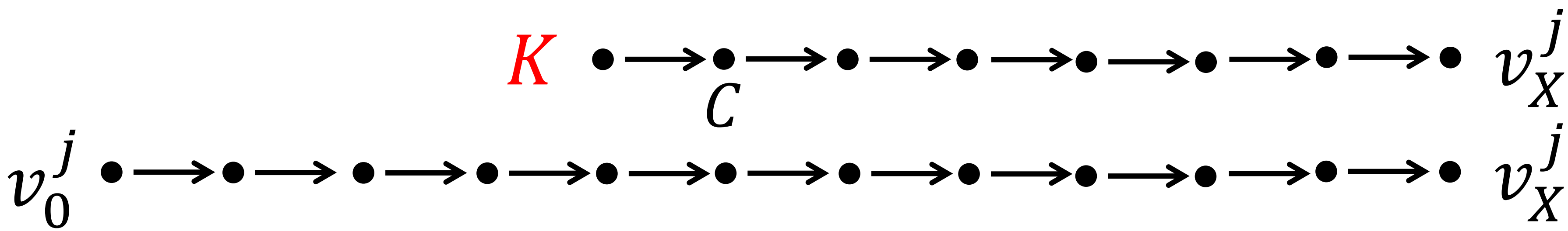


Online Phase

After user's key K is chosen:

- Query P to obtain C.
- Make a chain until it reaches one of end points (v_X^j) in T_{pre}.





K is one of the values in the matched chain.
(recovered with additional X steps)

Summary of Hellman's Tradeoff

Offline Phase:

$$(Time, Memory) = (2^n, M)$$

Online Phase:

$$(Time, Memory) = (X, negl)$$

Tradeoff:

$$Time = X = \frac{2^n}{Memory}$$

$$\Longleftrightarrow \boxed{Time \times Memory = 2^n}$$

Application to Key Recovery in HMAC/NMAC

A part of results in

Jian Guo, Thomas Peyrin, Yu Sasaki and Lei Wang, "*Updates on Generic Attacks against HMAC and NMAC.*" CRYPTO 2014.

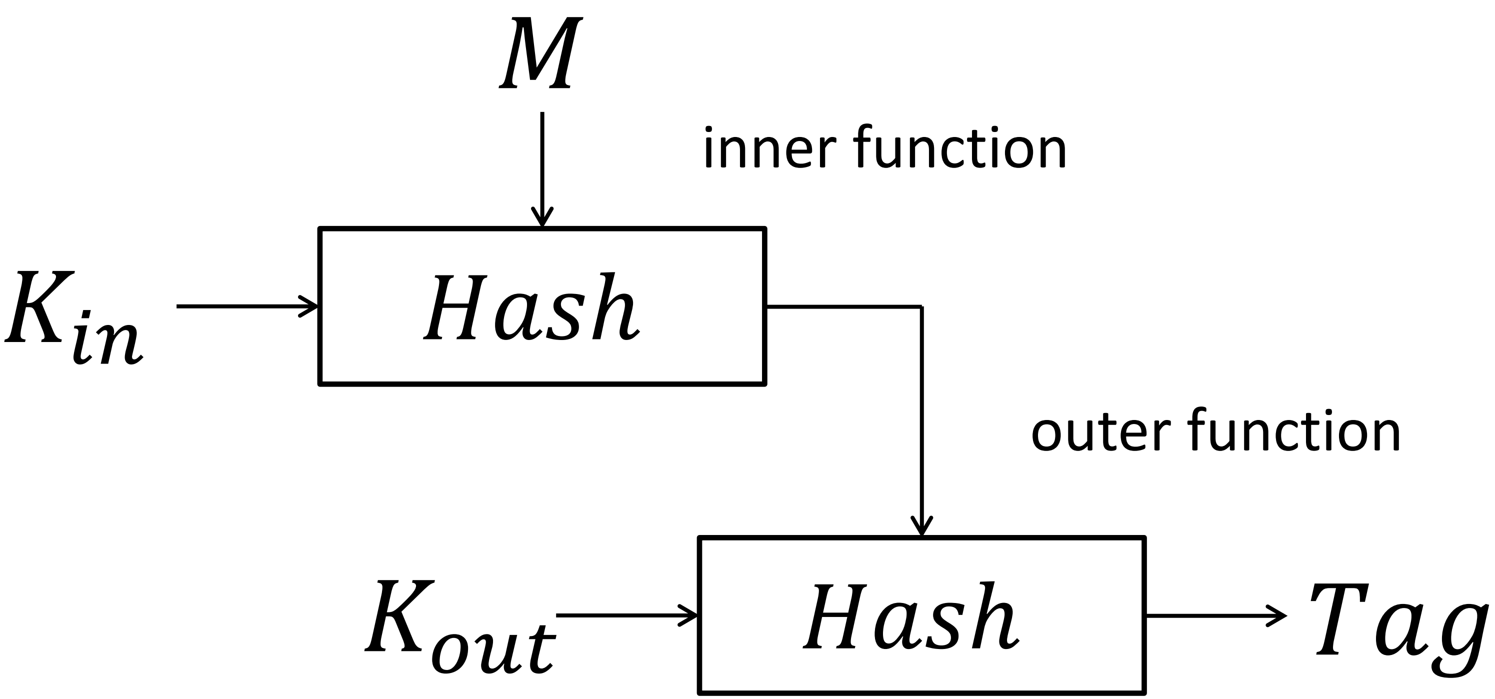
Hash Function based MAC

- NMAC (a base technique of HMAC)
  - Require 2 keys (inefficient)
  - Simple
- HMAC (widely used)
  - Require 1 key (practically efficient)
  - Complicated

For simplicity, NMAC is explained in this talk.

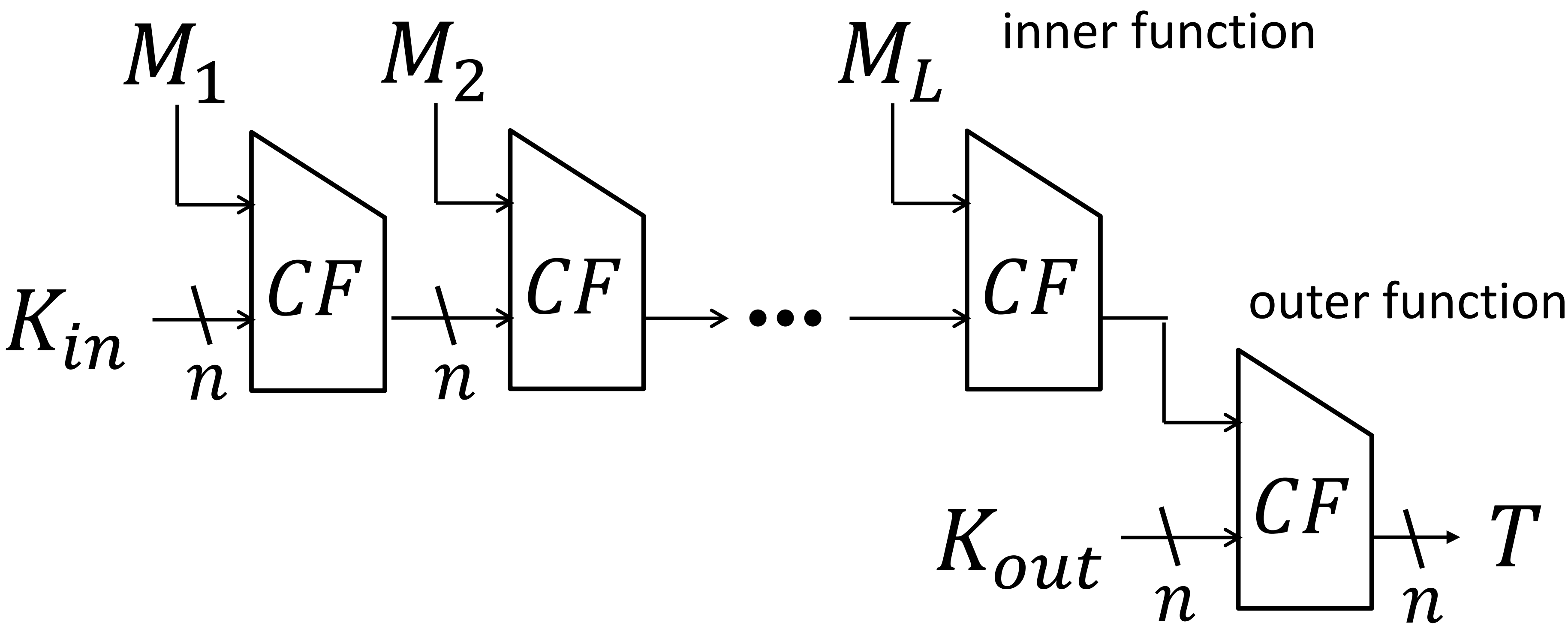
NMAC Specification

Two hash function calls by replacing IV with two keys K_{in} and K_{out}.



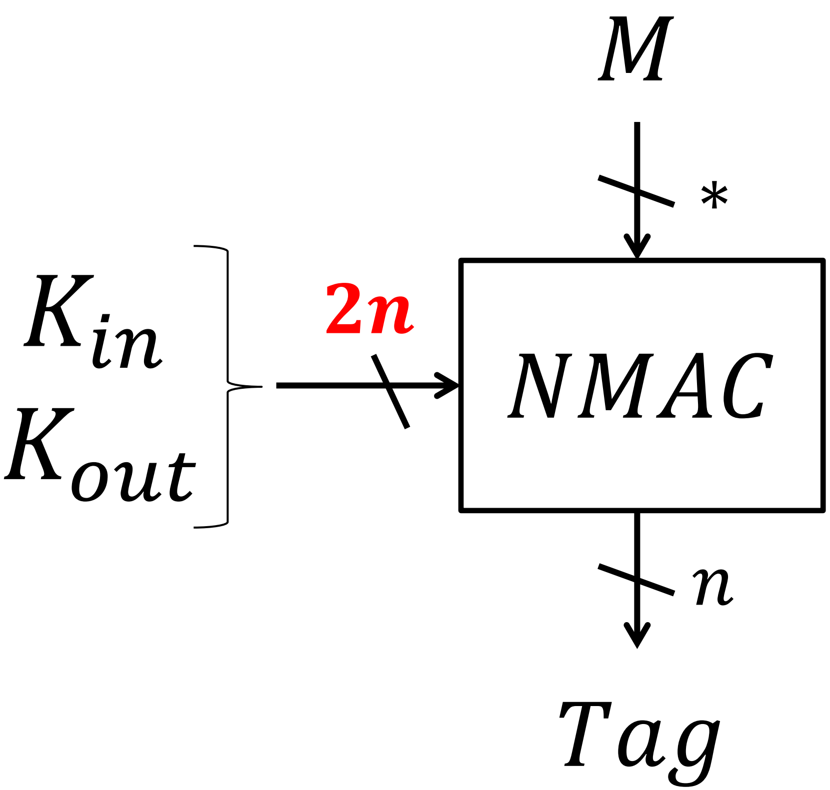
NMAC with Iterated Hash

Hash functions have some iterative structure, e.g. Merkle-Damgård structure



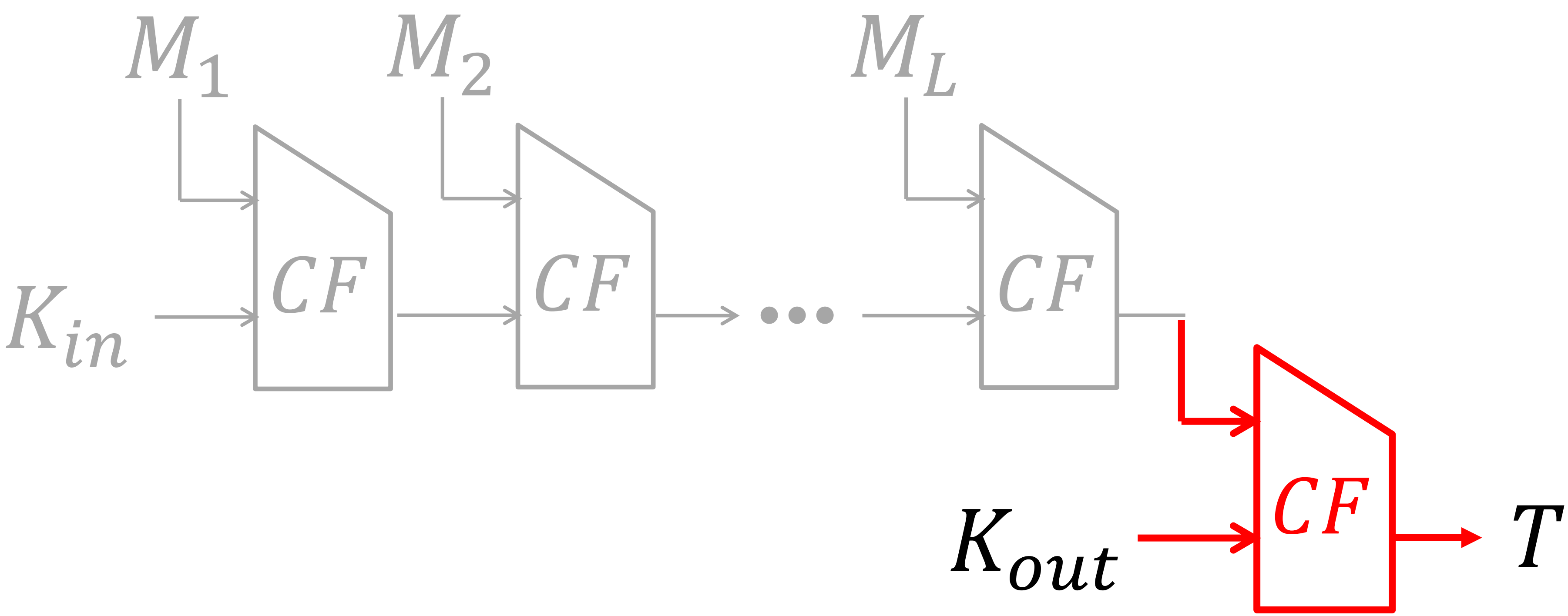
Straightforward Application

- Regard NMAC as $2n$-bit key primitive.
- Work in straightforward, but inefficient.



Divide-and-Conquer for K_{out}??

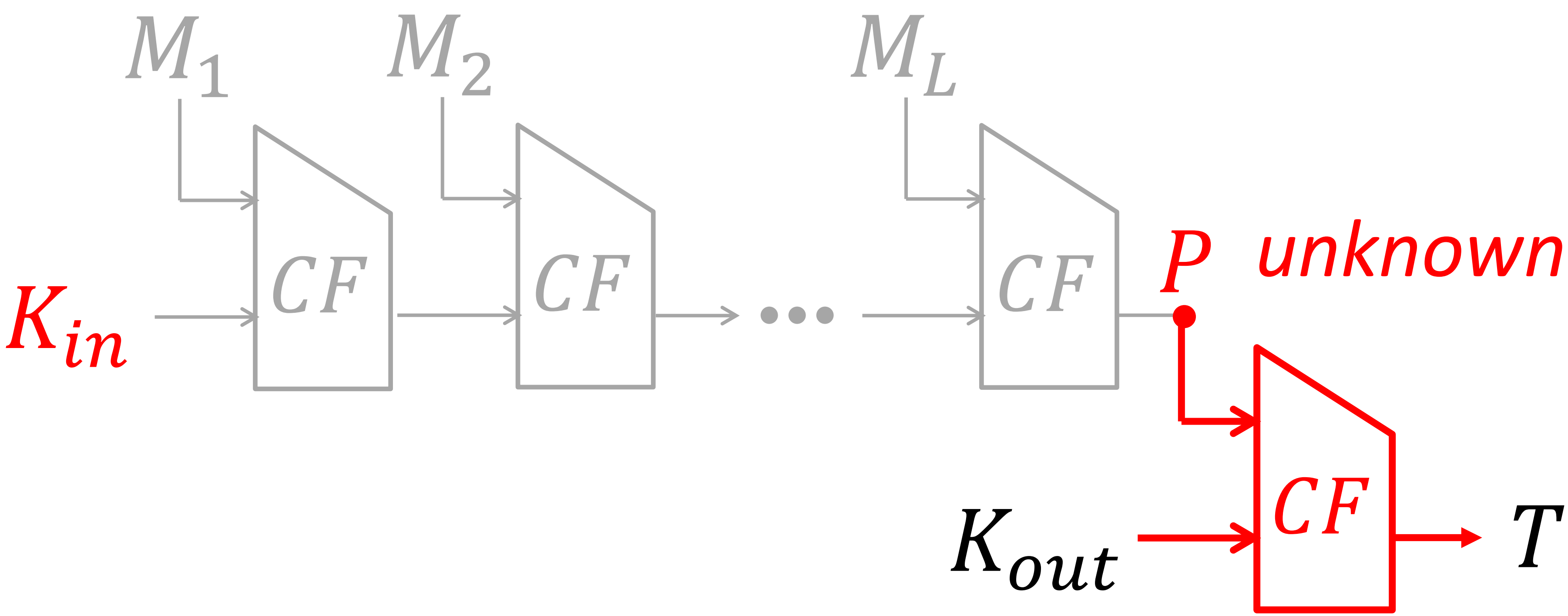
By focusing on outer function, K_{out} may be attacked independently from K_{in}.



Divide-and-Conquer for K_{out}??

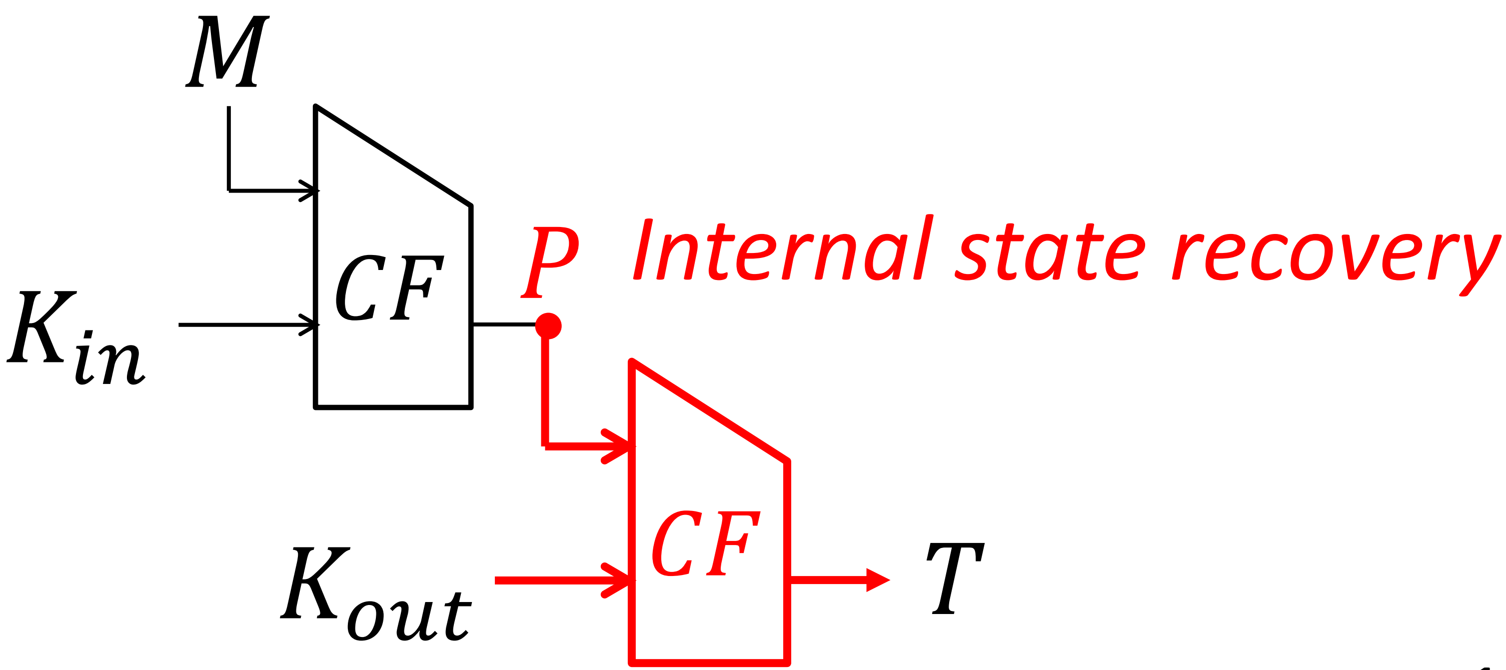
K_{in} hides the input value to outer function.

(simple application is impossible)



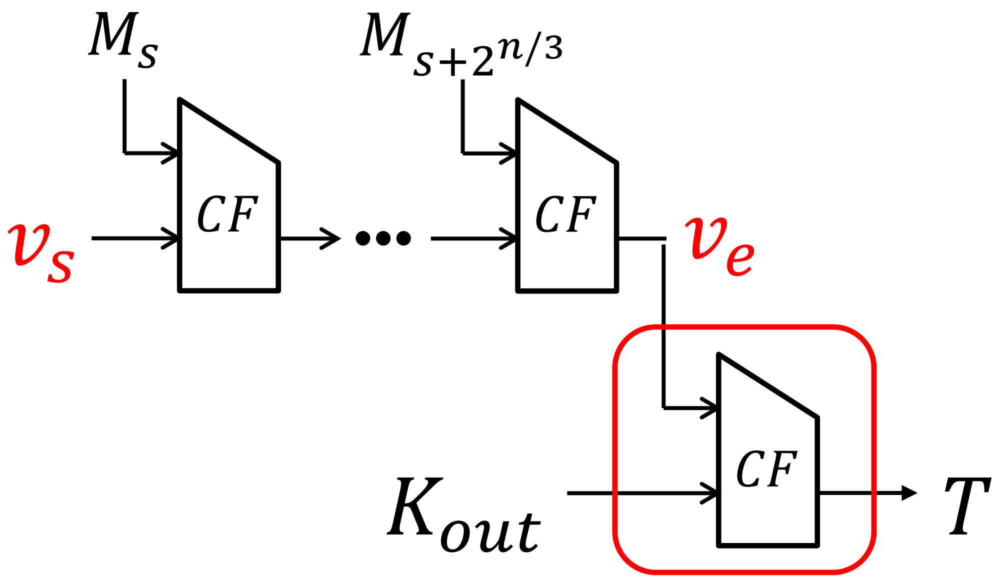
Internal State Recovery on NMAC

- [LPW14] recovers internal state P for some M.
- [LPW14] requires online queries.
- Hellman's tradeoff is meaningless without offline.



Our Method (Offline)

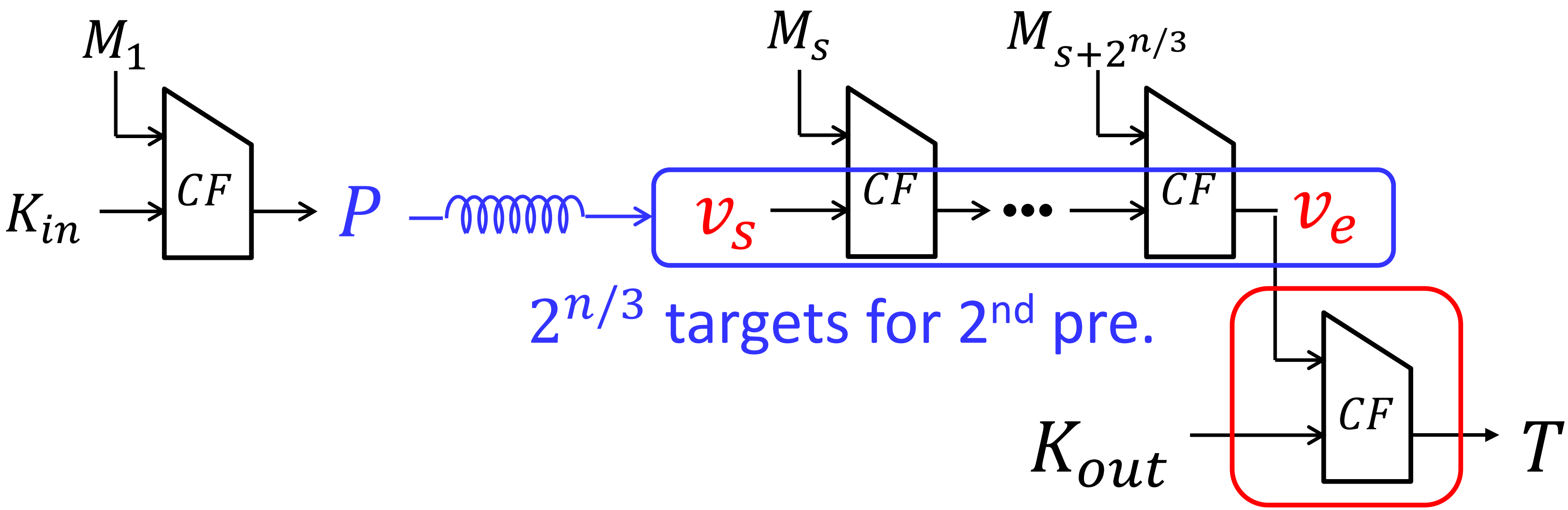
- Randomly choose v_s.
- Process v_s with $2^{n/3}$ blocks message to get v_e.
- Run Hellman's alg by assuming v_e is later obtained.



T_{pre} with ordinary Hellman's method

Our Method (Online)

- Recover internal state P with [LPW14].
- Run 2nd pre attack [KS05] from P to $2^{n/3}$ targets.
- Obtain T for v_e. Then, make a chain as usual.



Summary for Application to NMAC

- For MAC schemes, application of Hellman's tradeoff is non-trivial.
- By combining several existing techniques, application is still possible.
- For NMAC, we used
  1. Internal state recovery
  2. 2^{nd} preimage attack on Merkle-Damgård
  3. Hellman's time-memory tradeoff

Generalized Birthday Problem

A part of results in

Ivica Nikolić and Yu Sasaki, *"Refinements of the k-tree Algorithm for the Generalized Birthday Problem,"* Asiacrypt 2015, To appear.

Birthday Problem

$$F_1: \{0,1\}^* \rightarrow \{0,1\}^n$$
$$F_2: \{0,1\}^* \rightarrow \{0,1\}^n$$

Find input values (x_1, x_2) such that

$$F_1(x_1) \oplus F_2(x_2) = 0.$$

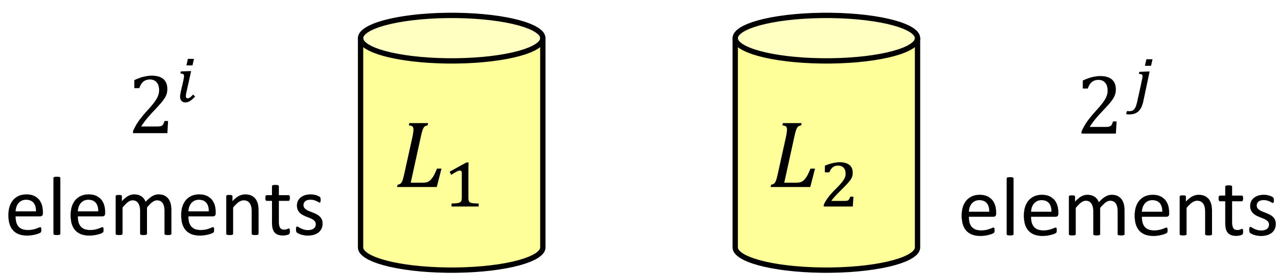
- can be defined for other group operations
- can be defined for an identical function but different input values

Solving Birthday Problem

Suppose that

- List L_1 contains 2^i pairs of $(x_i, F_1(x_i))$.
- List L_2 contains 2^j pairs of $(x_j, F_2(x_j))$.

When $2^{i+j} \geq 2^n$, solutions of $F_1(x_1) \oplus F_2(x_2) = 0$ exists with high probability.



Efficient Algorithm for Birthday Problem

For the birthday problem, several efficient algorithms can solve it with a complexity of

$$(Time, Memory) = (2^{n/2}, 2^{n/2}).$$

Moreover, with a cycle detection method:

$$(Time, Memory) = (O(2^{n/2}), negl)$$

Generalized Birthday Problem

$$F_1 : \{0,1\}^* \to \{0,1\}^n$$
$$F_2 : \{0,1\}^* \to \{0,1\}^n$$
$$\cdots$$
$$F_k : \{0,1\}^* \to \{0,1\}^n$$

Find a k-tuple input values $(x_1, x_2, \ldots x_k)$ such that

$$\bigoplus_{i=1}^{k} F_i(x_i) = 0.$$

Solving Generalized Birthday Problem

List L_i contains pairs of $(x, F_i(x))$.

When $|L_1| \times |L_2| \times \cdots \times |L_k| \geq 2^n$, a solution of generalized birthday problem exists with high probability.

It does not mean that the solution can be found with complexity $2^{n/k}$.

Wagner's k-Tree Algorithm [W02]

solves the problem for k with

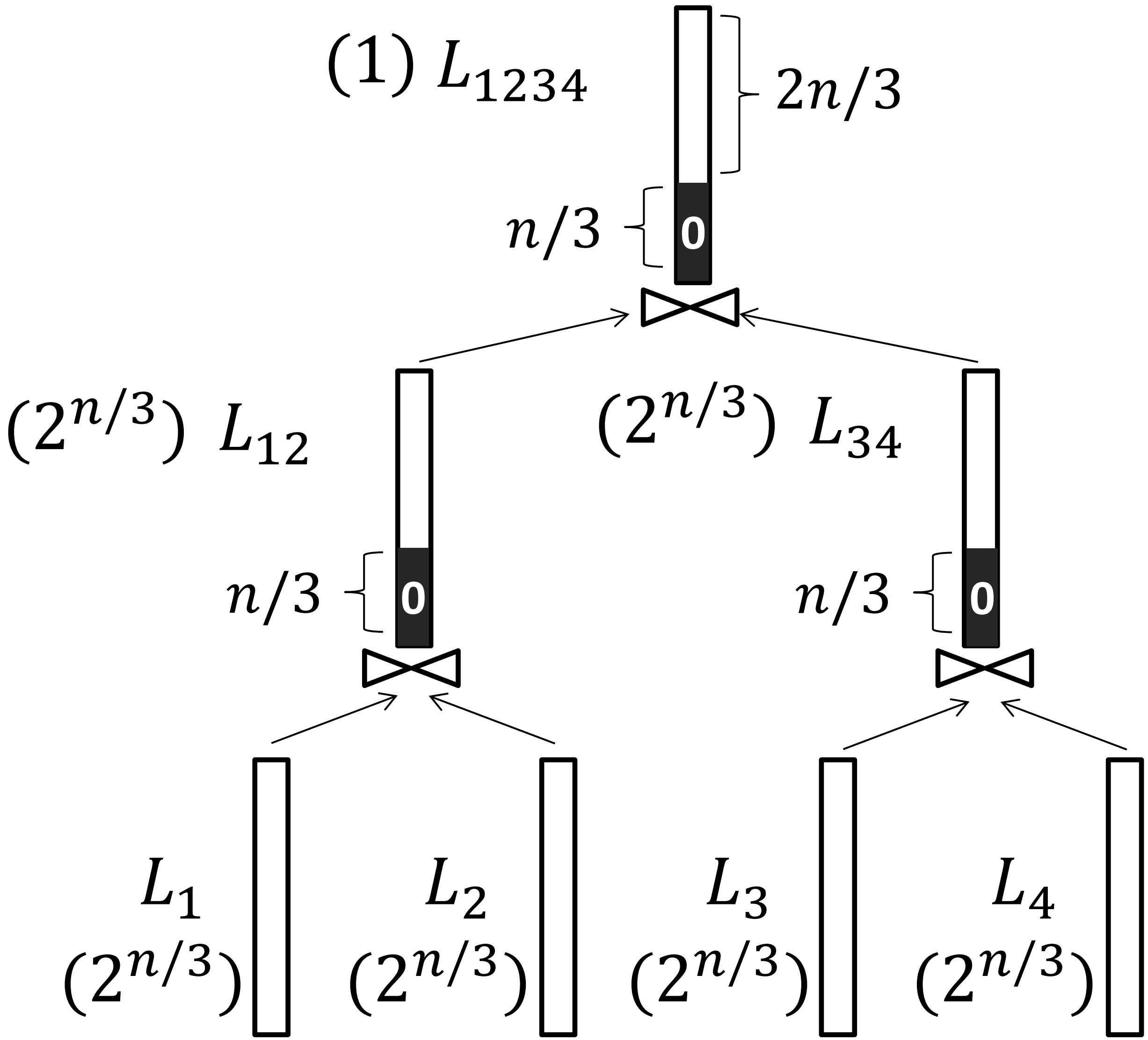
$$Time = Memory = 2^{\frac{n}{\lceil \log k \rceil + 1}}.$$

e.g.

- 4 lists $\rightarrow$ $k = 4$ $\rightarrow$ $T = M = 2^{n/3}$
- 8 lists $\rightarrow$ $k = 8$ $\rightarrow$ $T = M = 2^{n/4}$

Approach: divide-and-conquer

Example of k-Tree Algorithm ($k = 4$)



2nd layer:

Balance

$2n/3$ bits

1st layer:

Balance

$n/3$ bits

Introducing Time-Memory Tradeoff

- Memory is more costly than Time.
- E.g. $n = 160$ for SHA-1:
  - $2^{53.3}$ SHA-1 computations are feasible
  - $2^{53.3}$ memory seems hard (memory access is slow).

What's the best algorithm for the GBP with a small memory?

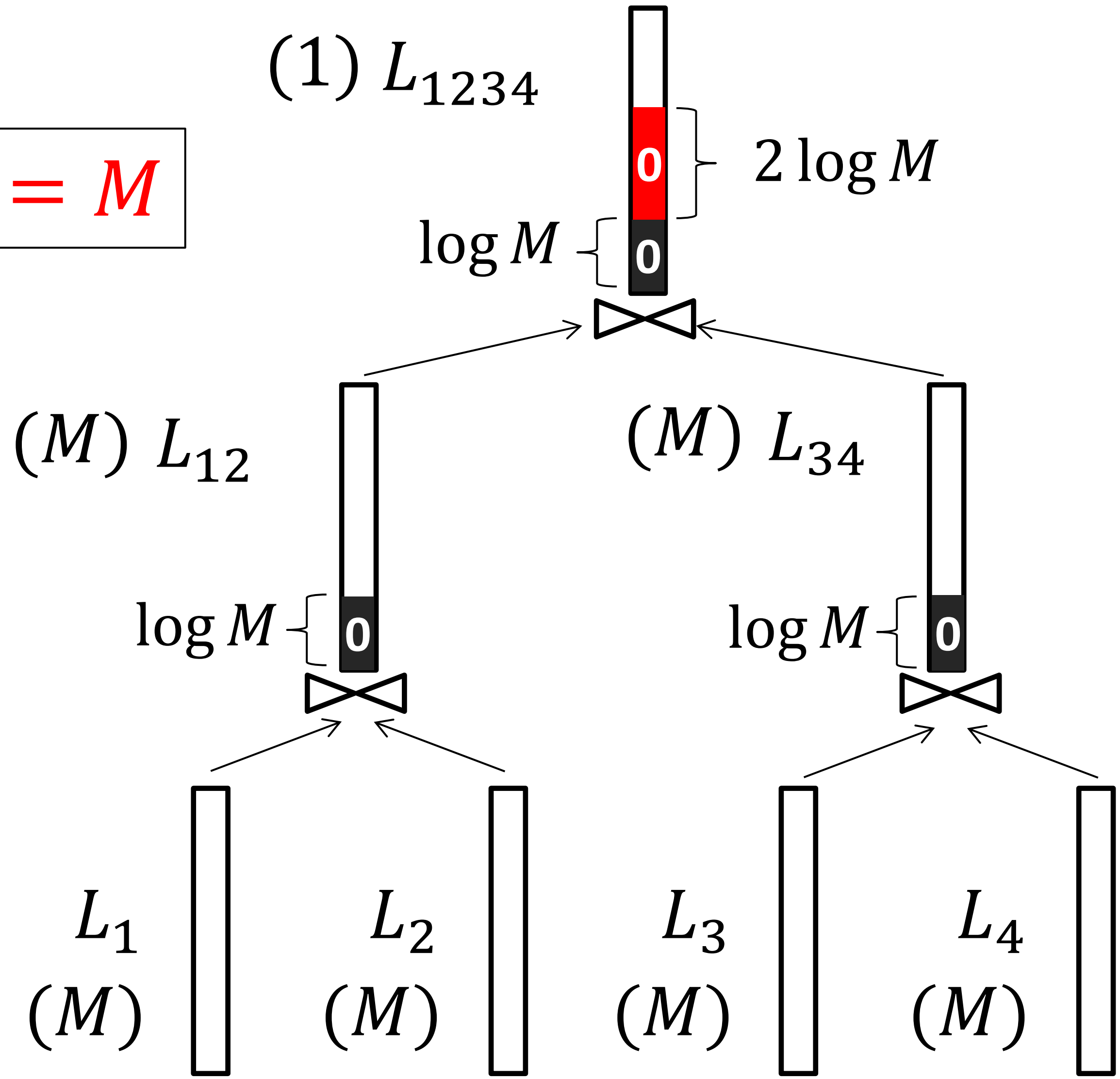
Previous Work

Not so many researches have been taken on the memory limited case of GBP

- D. J. Bernstein. “*Better price-performance ratios for generalized birthday attacks.*”, SHARCS'07
- D. J. Bernstein, T. Lange, R. Niederhagen, C. Peters, and P. Schwabe. “*FSBday.*“, Indocrypt 2009

How Does It Look Like?

$Time = M$



2nd layer:

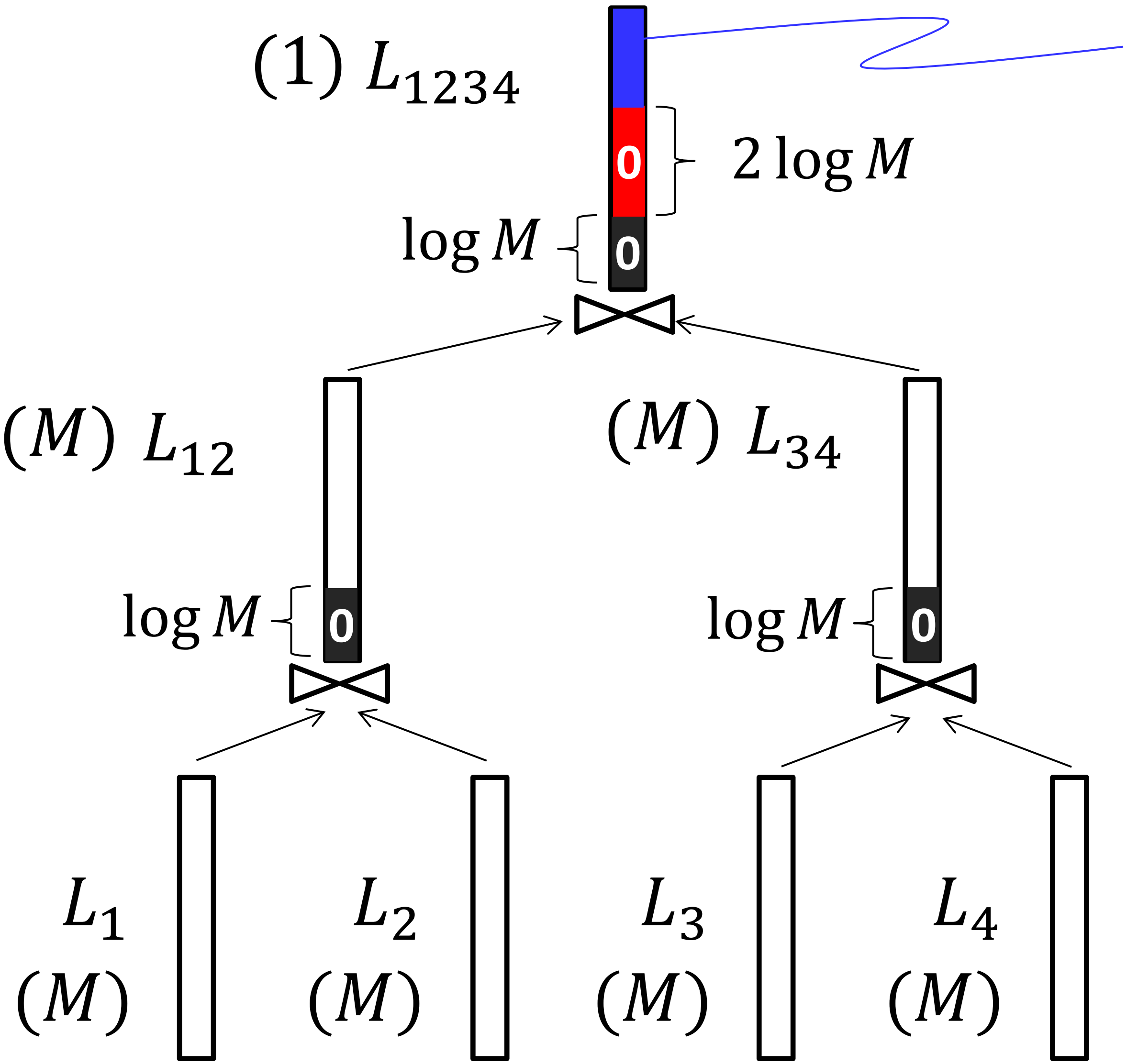
Cannot reach n bits

1st layer:

Cramp at least $\log M$ bits

Cannot store $2^{n/3}$

Previous Method 1



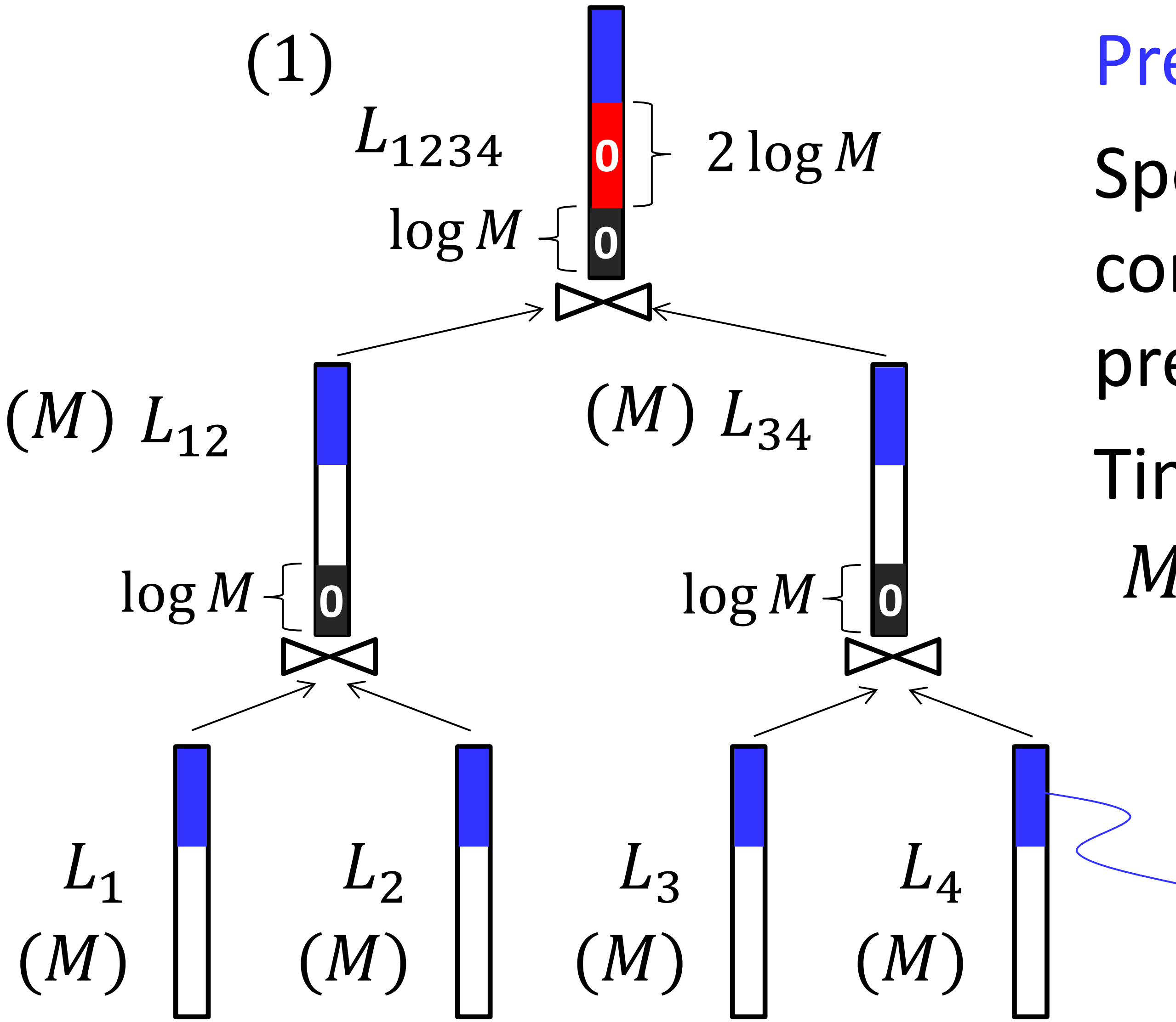
Simple Iteration:

Iterate until n bits become 0.

Time:

$M * 2^{n-3\log M}$

Previous Method 2



Prefilteration:

Spend some computation to prepare lists.

Time:

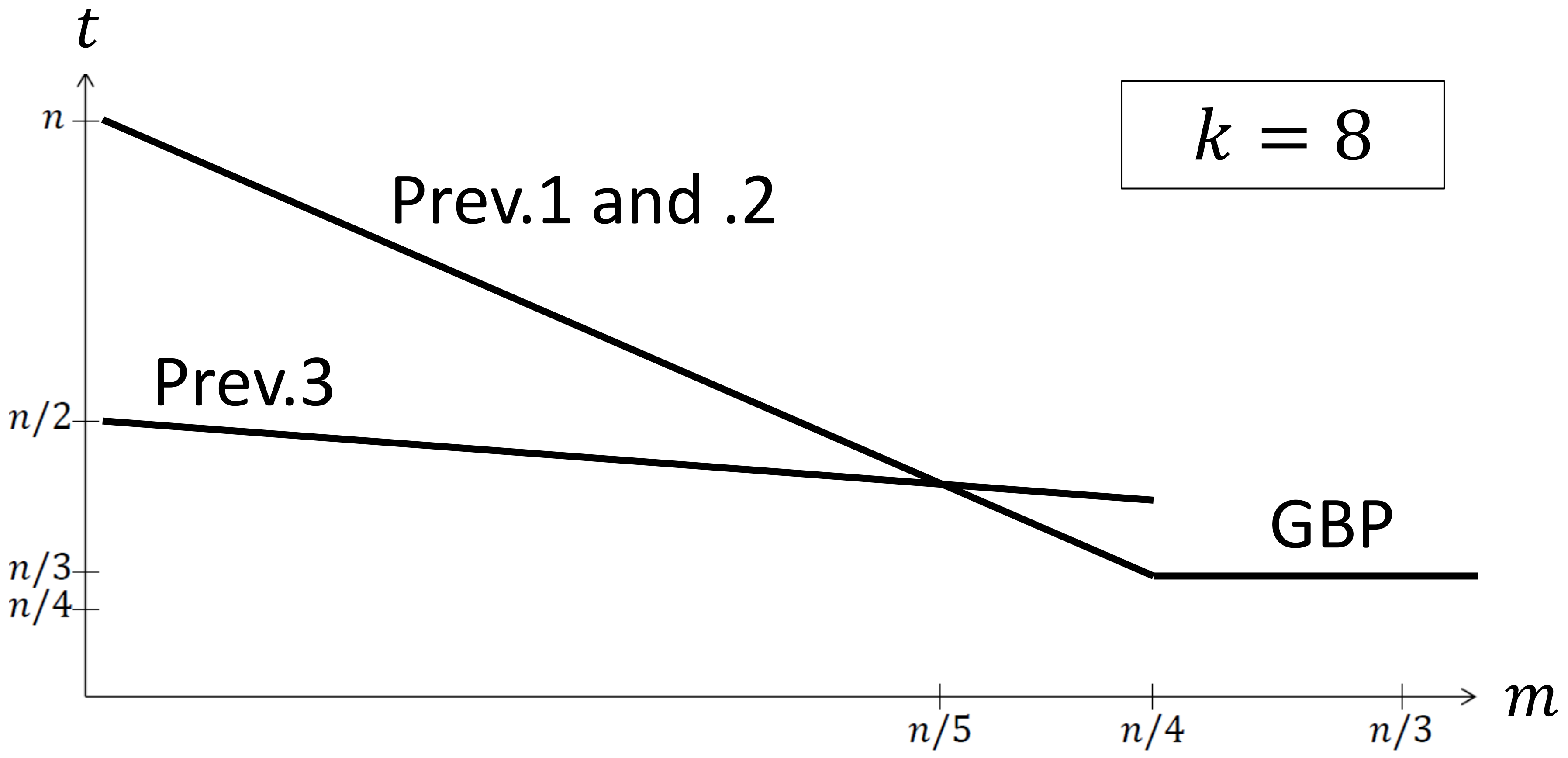
$M * 2^{n-3\log M}$

Previous Method 3

Only works when $f_1 = f_2, f_3 = f_4, \cdots$

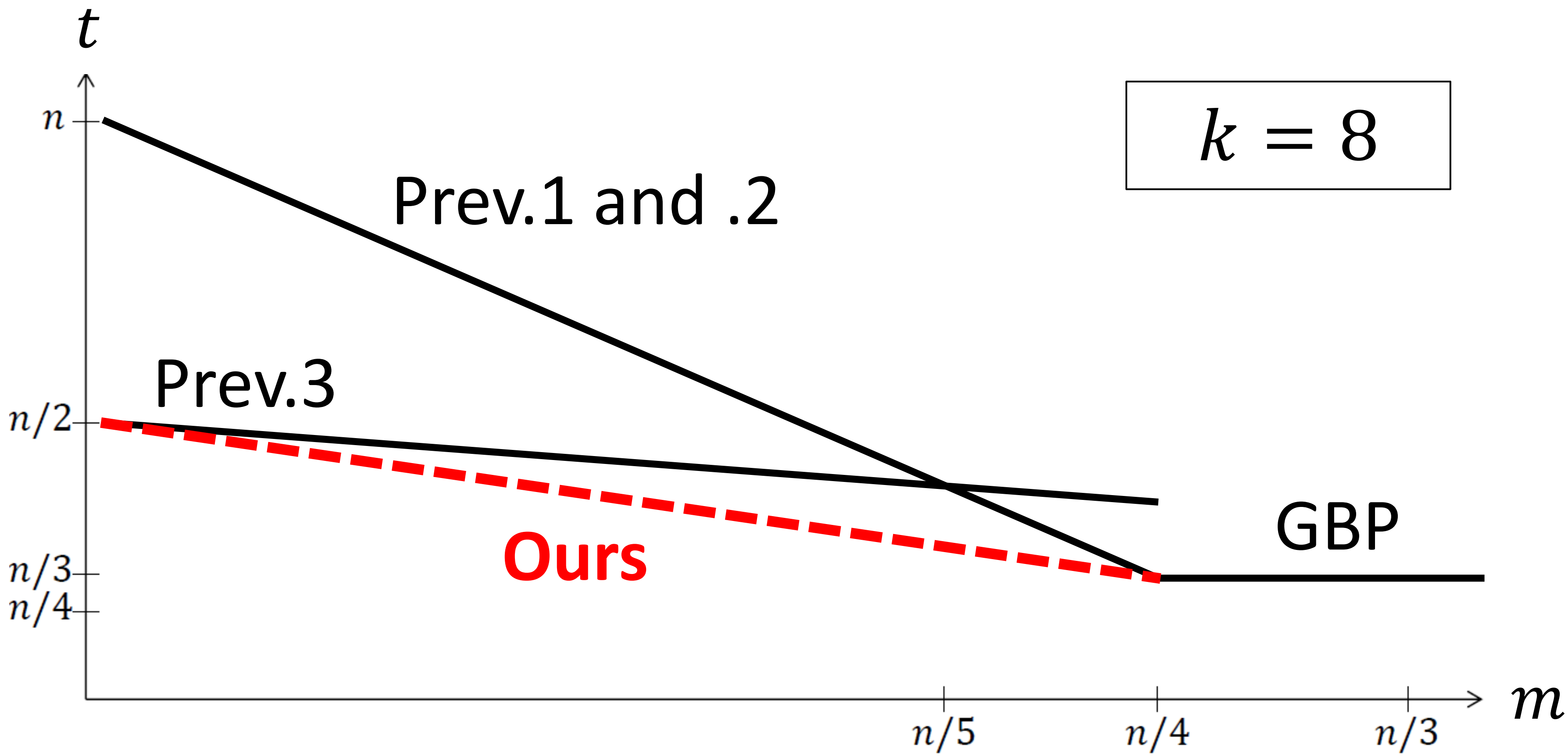
1. Run the k-tree algorithm for $f_1, f_3, f_5, \cdots$ with small M.
2. Run the k-tree algorithm for $f_2, f_4, f_6, \cdots$ with small M.
3. Run the memoryless collision search for the last merging phase.

Comparison of Previous Tradeoffs



- Prev.1 and .2 are good when m is relatively large.
- Prev.3 is opposite.

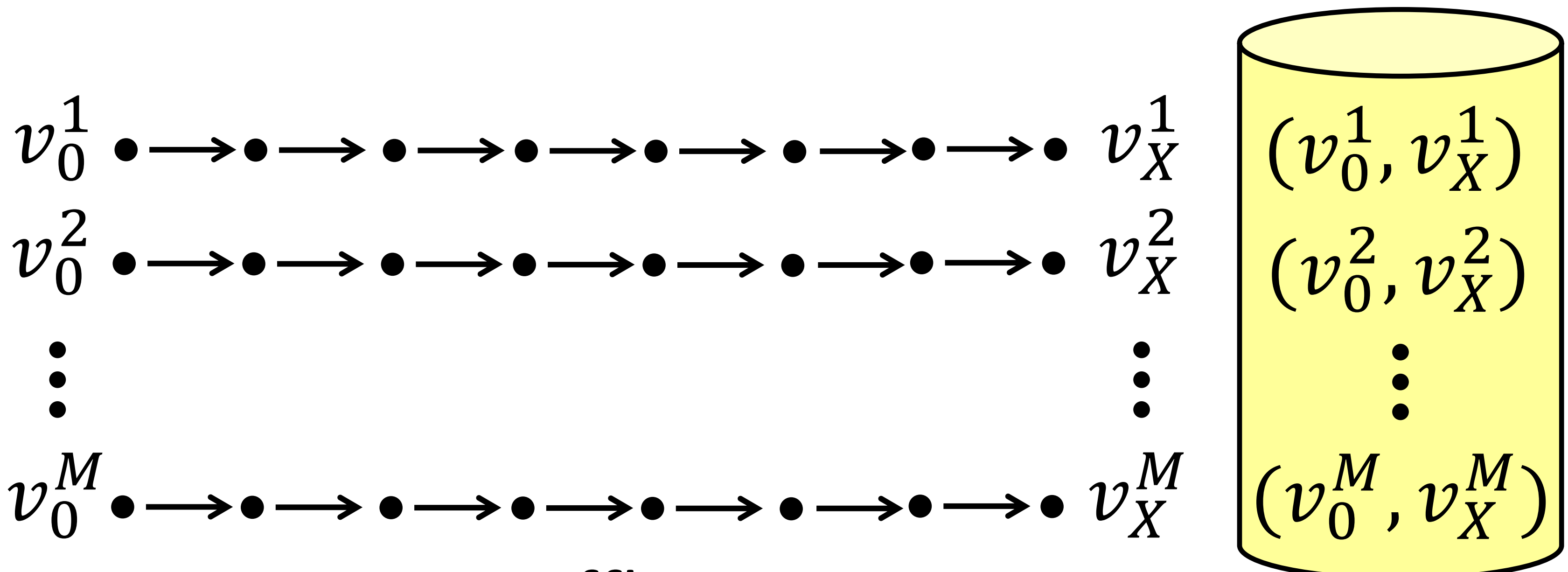
Our New Tradeoff



- take advantages of both methods
- only works when all f is identical

Hellman's Table for Public Functions

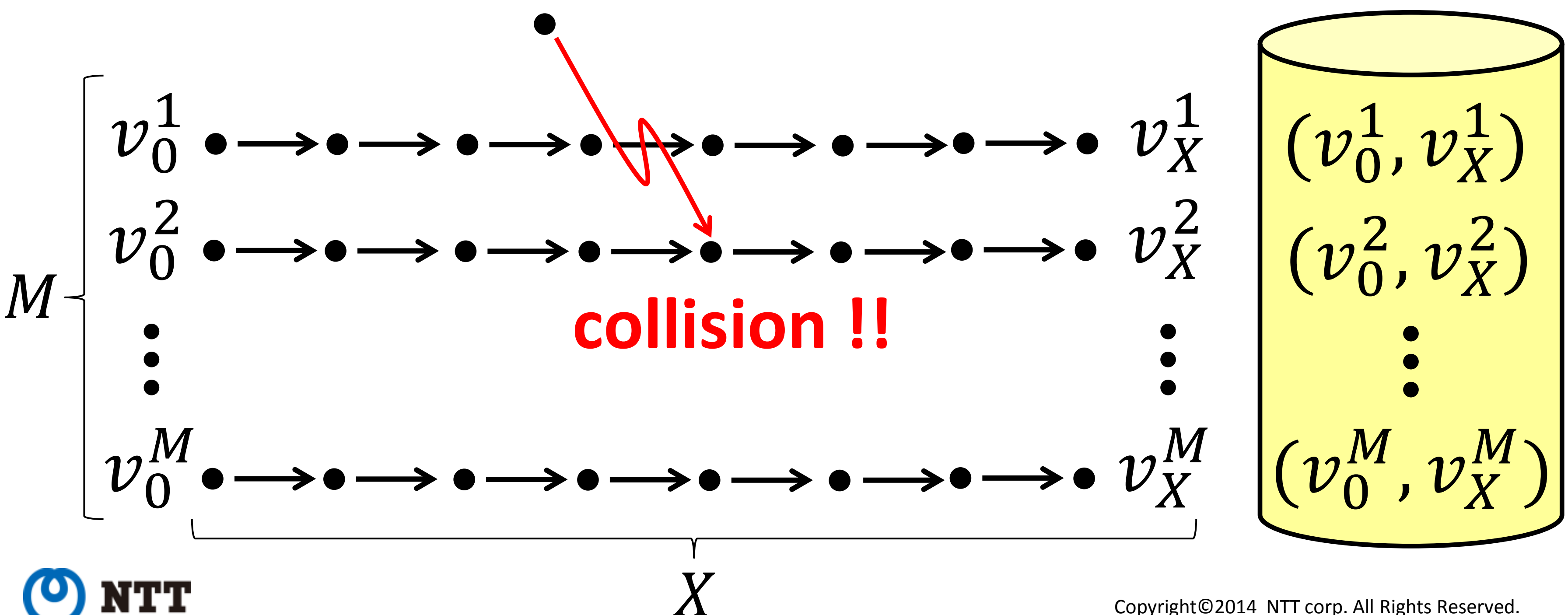
- Domain is infinite, impossible to examine all the input values.
- Identical idea, but different purpose.



Offline

Hellman's Table for Public Functions

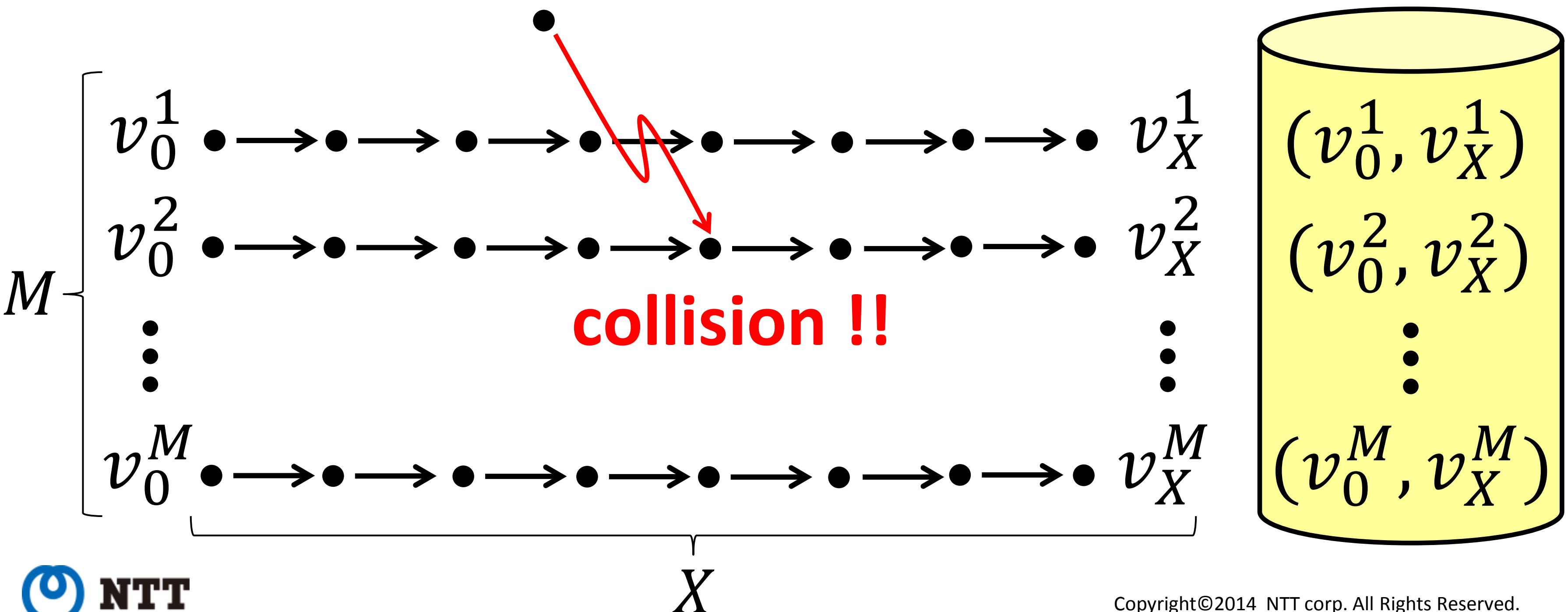
- Online phase of Hellman's algorithm generates a collision to one of the chains.
- Hellman's table is used for collision generation.



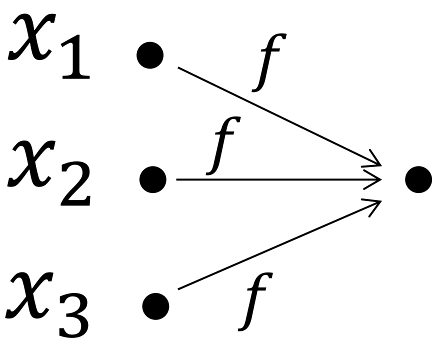
Hellman's Table for Public Functions

Fact 1 (Hellman's Table)

Once M chains of length X are computed, cost for generating collision is $O(\frac{N}{MX})$ per collision.



3-collision finding problem [JL09]



$$f(x_1) = f(x_2) = f(x_3)$$

- Well-known: $T = 2^{2n/3}$, $M = 2^{2n/3}$.
- [JL09]: $T = 2^{2n/3}$, $M = 2^{n/3}$

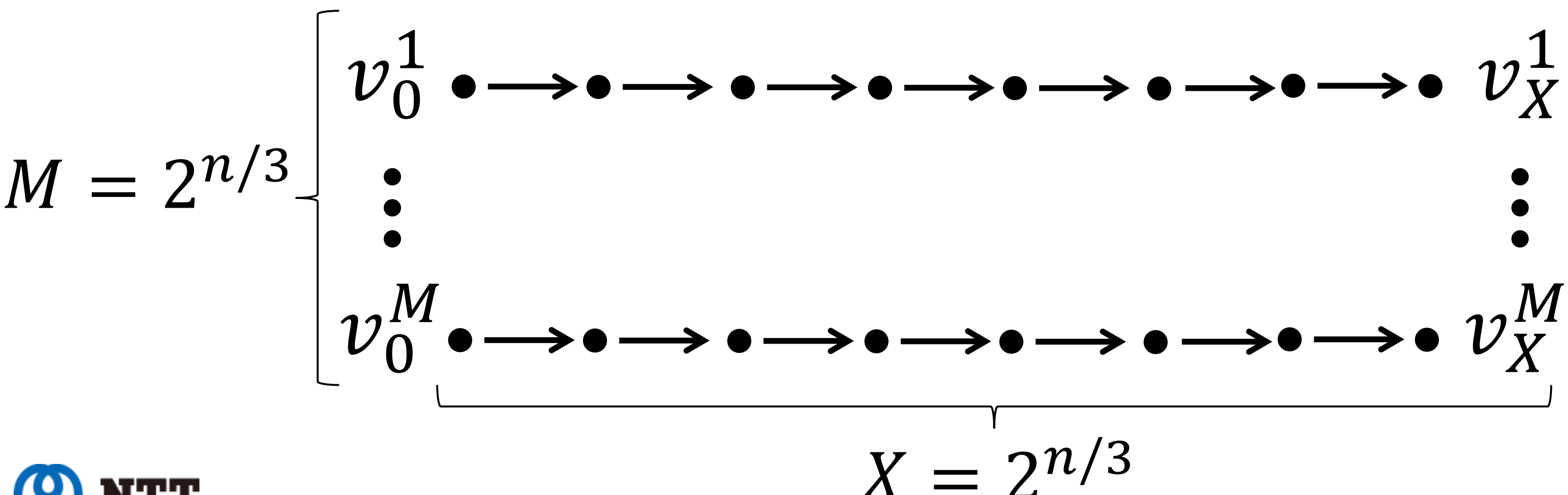
Previous Application of Hellman's Table

1. Generate chains: $T = 2^{2n/3}, M = 2^{n/3}$

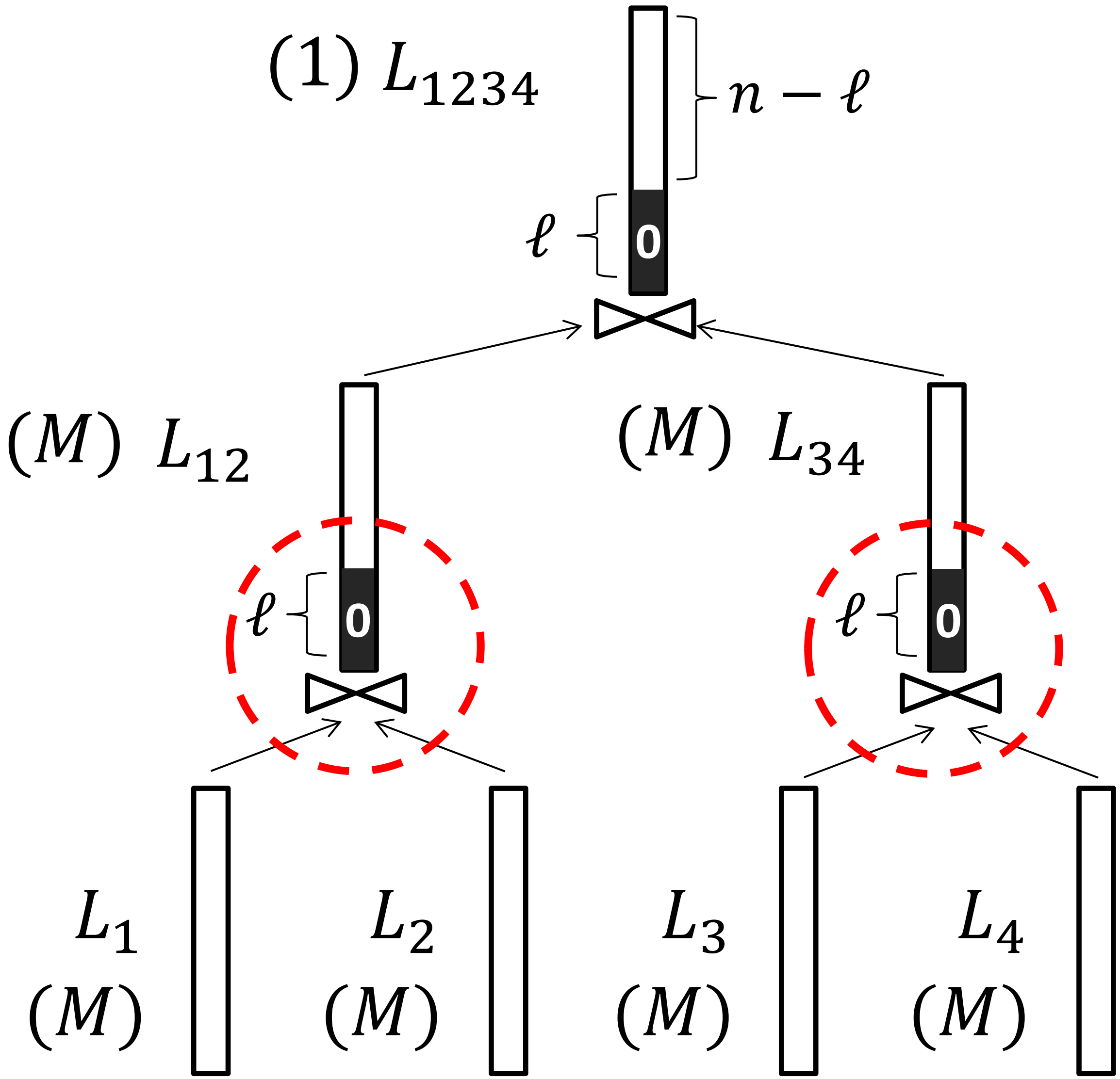
   Cost per collision becomes $O(2^{n/3})$.

2. Generate $2^{3/n}$ collisions: $T = 2^{2n/3}, M = 2^{n/3}$
3. Generate $2^{2n/3}$ values : $T = 2^{2n/3}, M = negl$

coll



Hellman's Table Fits k-Tree

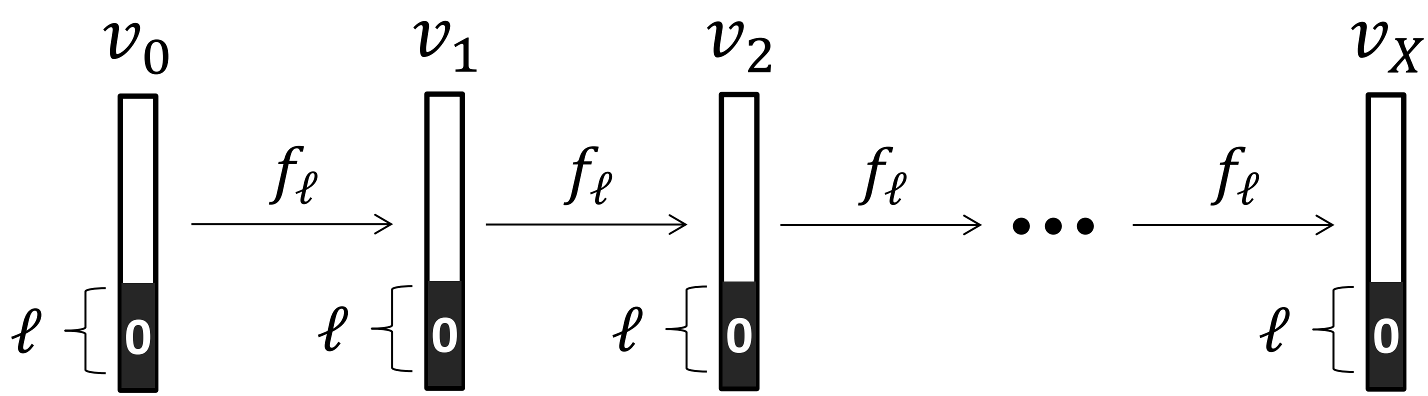


- 1st layer of k-tree algorithm generates many ***partial*** collisions.

- Suitable for Hellman's table.

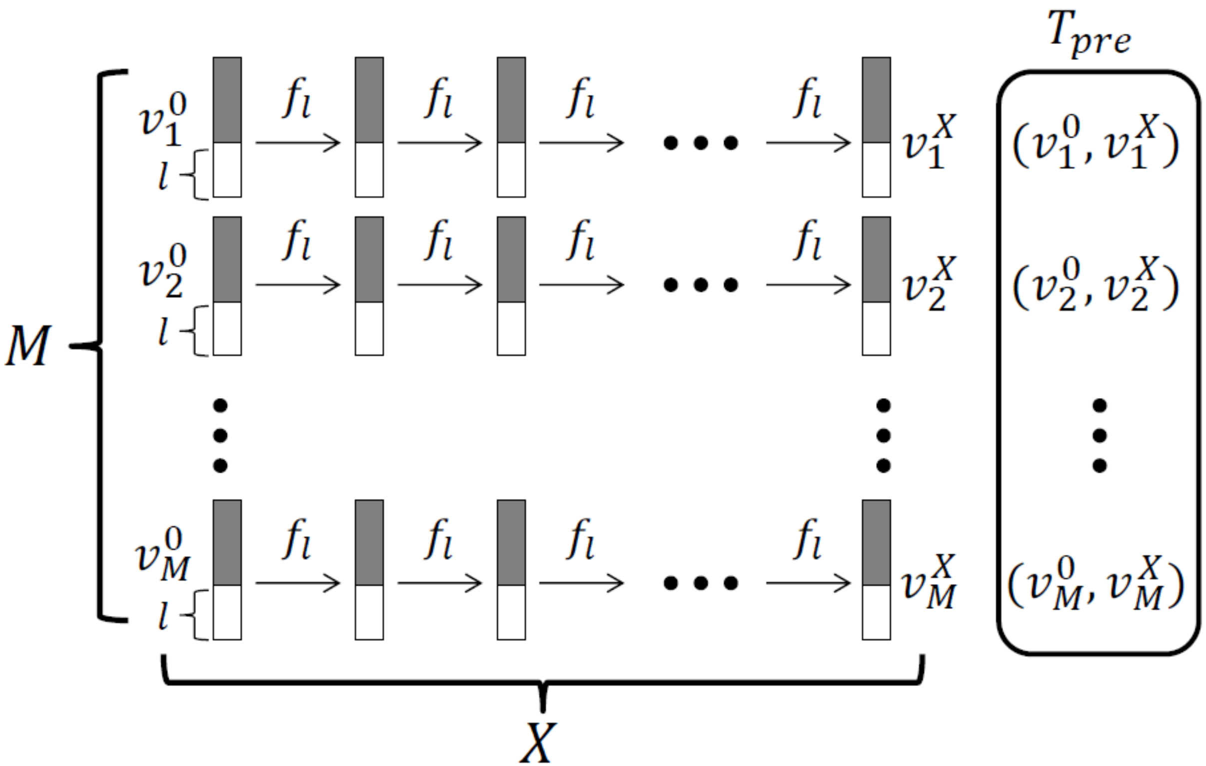
Reduction Function

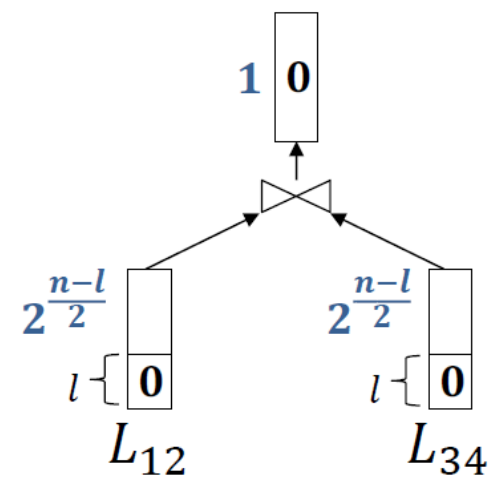
- Ordinary Hellman's table detects collisions instead of partial collisions.
- The k-tree alg finds partial collisions (otherwise divide-and-conquer doesn't work).
- Reduction function f_ℓ discards $n - \ell$ MSBs and only uses ℓ LSBs for building chains.



Our Algorithm for k-Tree

1. Construct Hellman's table.
2. Generate $2^{\frac{n-\ell}{2}}$ ℓ-bit collisions for L_{12} and L_{34}.
3. Find a collision on $n-\ell$ bits between L_{12} and L_{34}.





Complexity Evaluation

Step 1: $Time = MX, \quad Memory = M$

Step 2: $Time = 2^{\frac{n+\ell}{2}}/MX, \quad Memory = 2^{\frac{n-\ell}{2}}$

Step 3: $Time = 2^{\frac{n-\ell}{2}}, \quad Memory = negl$

Balance all the Steps:

$$T^2M = N$$

Our Algorithm for General k

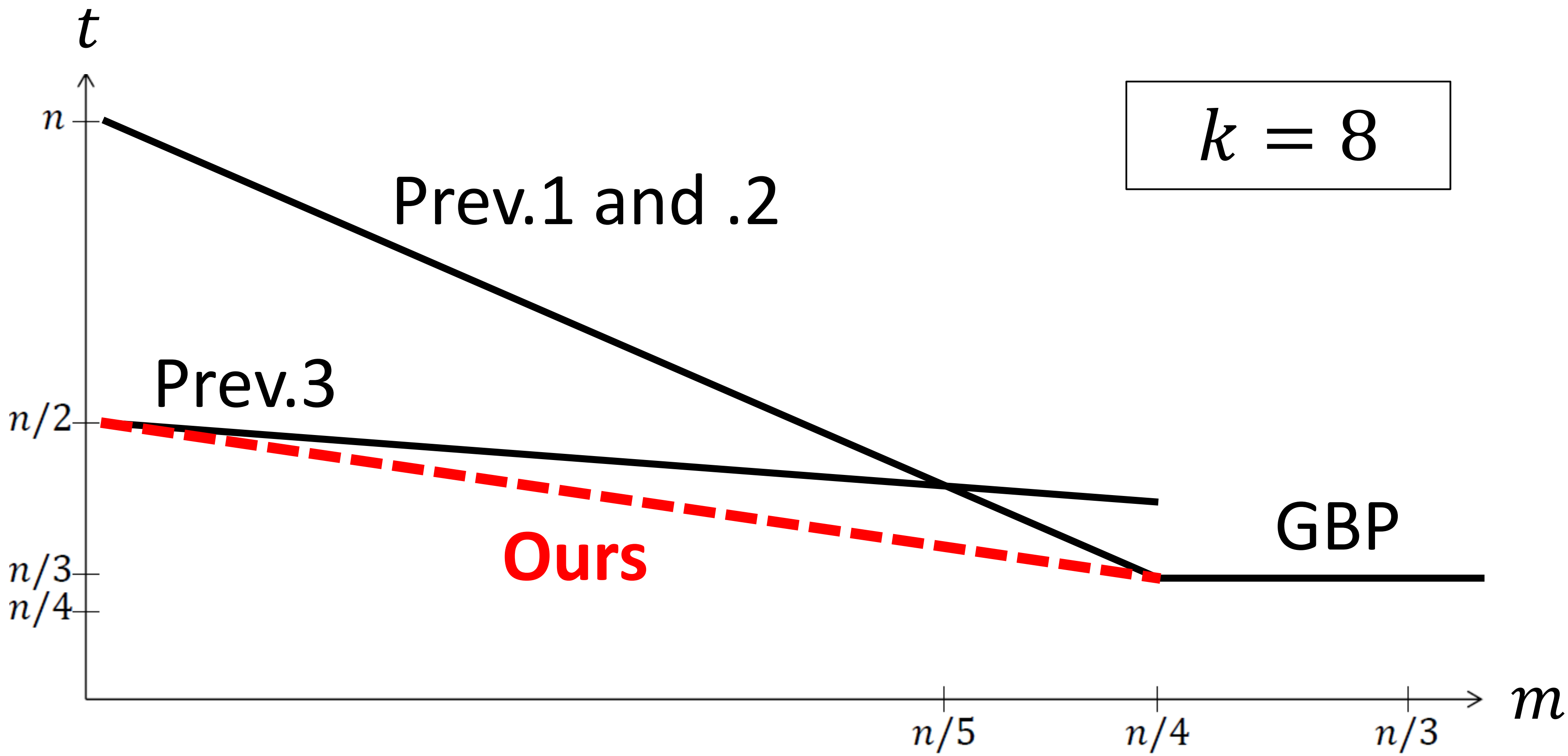
- Partial collisions in the first layer are always generated with Hellman's table.

$$T^2 \cdot M^{\log k - 1} = N$$

- Example (k=8):

Method	Curve	M	T
Prev work 1	$TM^3 = N$	$2^{n/6}$	$2^{6n/12}$
Prev work 3	$T^2M = N$	$2^{n/6}$	$2^{5n/12}$
Ours	$T^2M^2 = N$	$2^{n/6}$	$2^{4n/12}$

Our New Results



- take advantages of both methods
- only works when all f is identical

Concluding Remarks

Conclusion

Recent results using Hellman's tradeoff

- Secret function
  - Outside construction makes application non-trivial
  - K_{out} recovery in NMAC/HMAC
- Public function
  - Useful when many collisions are generated
  - New time-memory tradeoff for GBP

Thank you for your attention !!