Efficient Message Authentication Codes with Combinatorial Group Testing

Kazuhiko Minematsu (NEC Corporation)

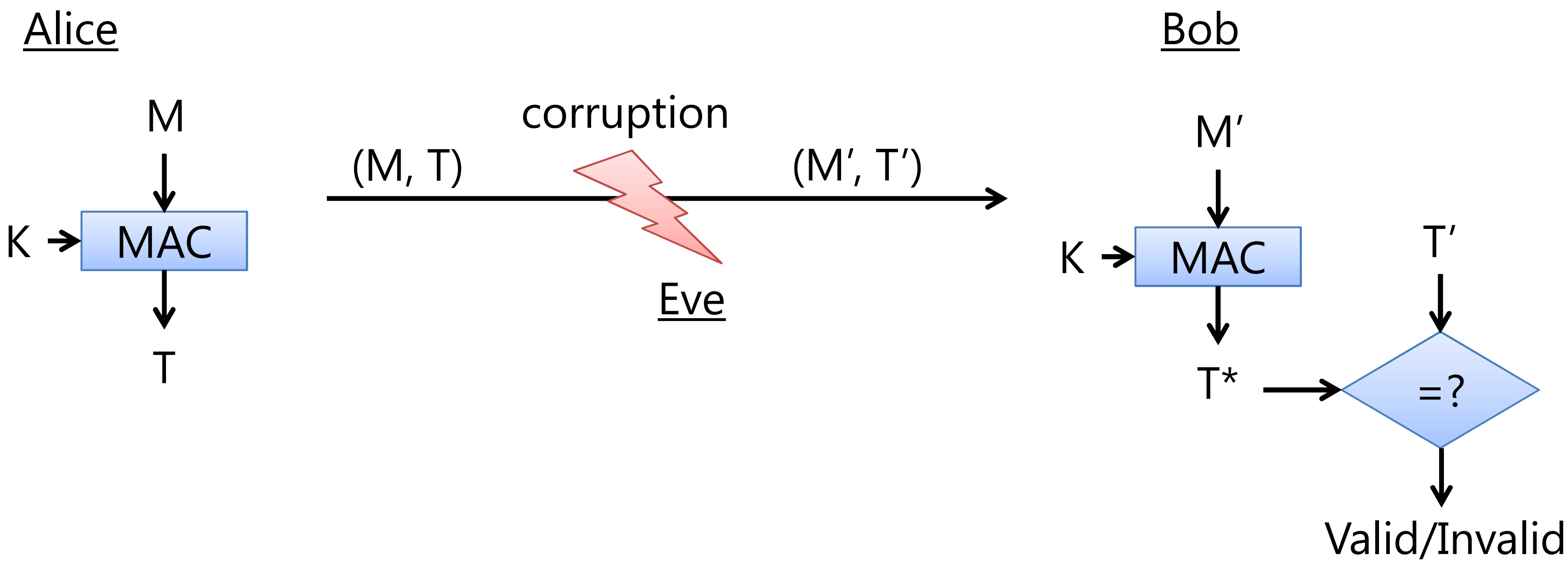
The paper was presented at ESORICS 2015,
September 23-25, Vienna, Austria

ASK 2015, October 3, Singapore

Introduction

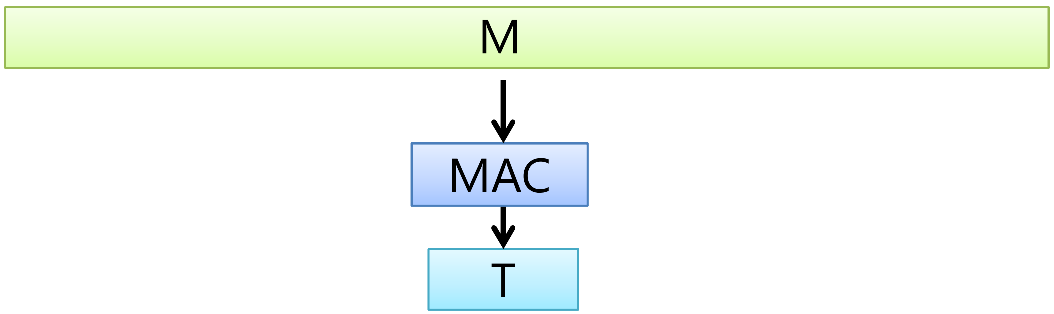
Message Authentication Code (MAC)

- Symmetric-key primitive to detect forgery
- Compute T = MAC(K,M), send (M,T)
- Receiver checks if tag is correct using the same K
- Known efficient constructions, e.g. CMAC and HMAC



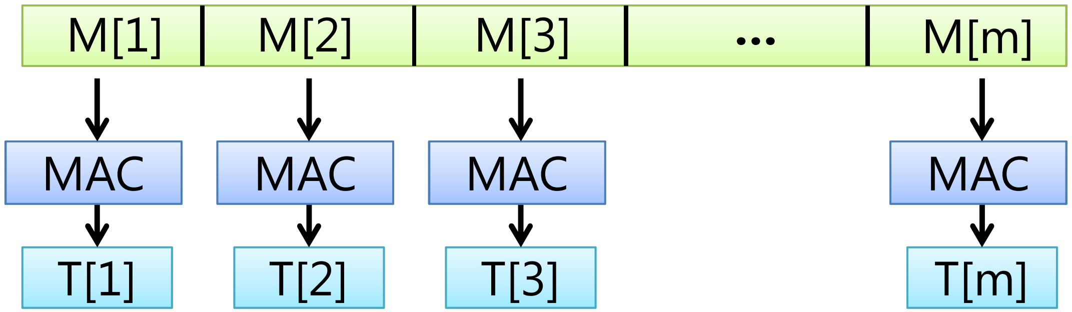
Limitation of standard MAC

- Verification result is binary : when verification fails, no information beyond the existence of corruption
  - HDD sectors, File sections, DB entries…
- If we know which parts have been corrupted, it would be useful to reducing cost, e.g.
  - retransmission in communication network
  - manual investigation in digital forensics
- Allows "fuzzy" authentication



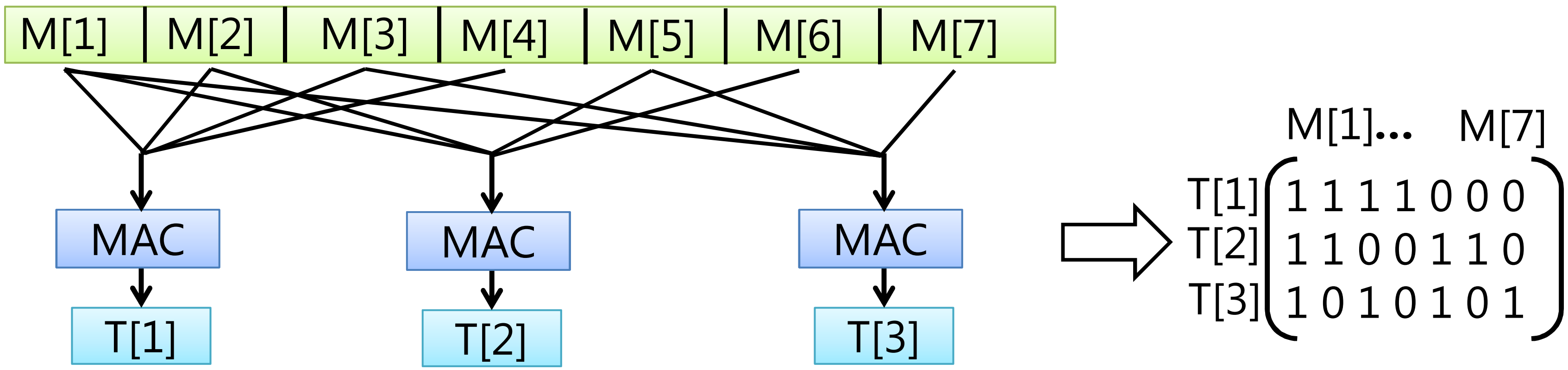
Finding corruptions

- Trivial solution : taking multiple tags for individual parts (data items)
- We can always identify all corrupted items, but tags impact storage
- Tread-off between the quality of information and storage : could it be improved?



Better tread-off

- A promising direction is taking multiple tags for *overlapping* subsequences of items
- Example: for 7 items, take 3 tags for (M[1],M[2],M[3],M[4]), (M[1],M[2],M[5],M[6]), and (M[1],M[3],M[5],M[7])
- Represented as a 3x7 binary matrix



Better tread-off

- Verification result is a 3-bit vector
  - "1" denotes the (index of) unmatched tag string
- Uniquely mapped to the index of single corrupted item, or no corruption
- That is, if at most 1 item is corrupted, this scheme can identify it

	M[1]	...					M[7]
T[1]	1	1	1	1	0	0	0
T[2]	1	1	0	0	1	1	0
T[3]	1	0	1	0	1	0	1

E.g. (011) implies M[1] to M[4] are uncorrupted & only M[5] can affect both T[2] and T[3]

Verification Result	000,	001,	010,	011,	100,	101,	110,	111
Index of corrupted item	none,	7,	6,	5,	4 ,	3,	2,	1

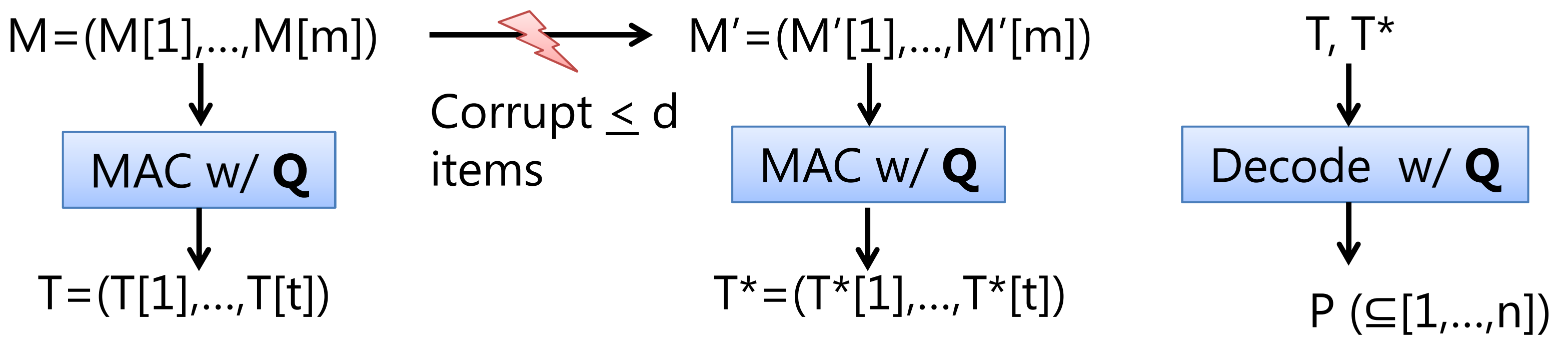
Combinatorial group testing (CGT)

Combinatorial group testing (CGT)

- What we are doing is an application of combinatorial group testing (CGT)
- CGT : a method to identify defectives via group test (“does group A contain any defective ?”)
  - Introduced by Dorfman during WWII (1943), as a method to effectively find bad blood supplies
  - Widely applied to biology and information science (see [Du-Hwang 00])
- In our case,
  - group test = tag check
  - Defective = corrupted item
  - Tags are non-adaptively computed – non-adaptive CGT (NCGT)

Problem setting

1. We have a list of data items, M=(M[1],…,M[m]), and (t x m) binary test matrix, **Q**
   (each M[i] is a bit string)
2. We take a tag vector, T = (T[1],…,T[t]), following **Q**
3. An adversary *A* corrupts at most d items
   (M,T) => (M',T)
4. At verification, we take local tag vector T*=(T*[1],…,T*[t]) for M' and check if T*[i] = T[i] for all i
5. Evict all items in negative tests (valid tags)
   - if T*[i] = T[i], then evict all j s.t. $\mathbf{Q}_{i,j}=1$
   - aka *naïve decoder* in CGT
6. Outputs indexes of all remaining items as corrupted



Building Test Matrix

- Then, how we build (t x m) binary test matrix **Q** ?

- For making this scheme to work, **Q** must be d-disjunct
  - Any union (bitwise OR) of $\leq$d columns of **Q** does not cover another column of **Q**
- d-disjunct matrix
  - extensively studied from combinatorics and coding theory
- For given m and d, $t = O(d^2 \log m)$
  - Classical methods w/ larger order (e.g. [Macula 96])
  - Matching deterministic method [Porat-Rothschild 08]
- We will not go further here

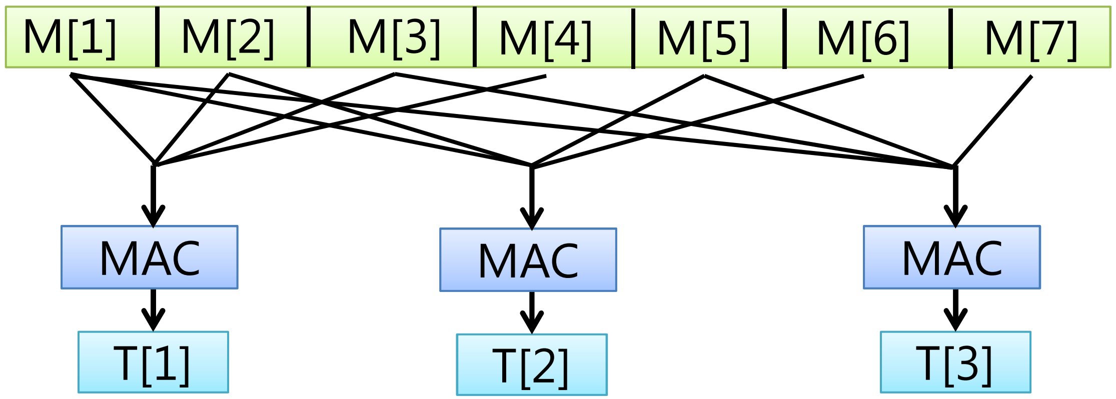
Previous works

- MAC/hashing/signature combined with CGT has been proposed and studied in various contexts
- MAC : [Crescenzo-Arce 04] [Goodrich-Atallah-Tamassia 05] etc.
- Hashing : Corrupltion-localizing hashing [Crescenzo-Jiang-Safavi-Naini 09], [Bonis-Crescenzo 11] etc.
- Signature : Batch signature verification [Zaverucha-Stinson 09]
- Applied to data forensics, computer virus detection, HDD integrity check, etc.

Efficient MAC with CGT

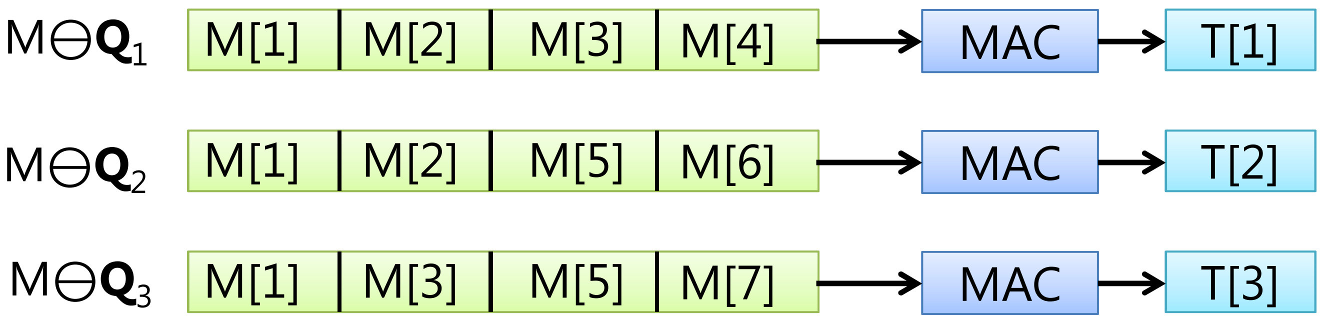
Motivation

- Storage cost is reduced from O(m) to O(d^2log m), if we use optimum **Q**
- How about computation cost ?
  - In standard MACs, taking single tag needs O(m) computation, assuming item processing as unit computation
  - (To the best of our knowledge) not studied in the previous works
    - the underlying MAC or hash is treated as a black box



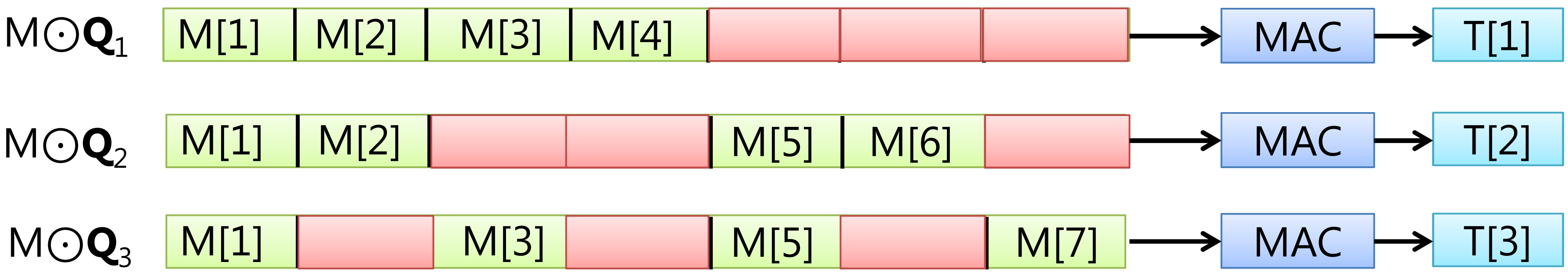
Naïve view

- Let $\{0,1\}^{*m}$ be the (normal) vector space of m-strings
  - Each string is a non-empty bit sequence of any length
- For M in $\{0,1\}^{*m}$, let $M \ominus \mathbf{Q}_i$ be the extracted subsequence of M for $\mathbf{Q}_i$ (i-th row of **Q**)
  - E.g. (M[1],M[2],M[3]) $\ominus$ (1,0,1) = (M[1],M[3])
- Naïve MAC w/ CGT method : T[i] = MAC($M \ominus \mathbf{Q}_i$)
  - O(Hw(**Q**)) = O(mt) computation, usually >> O(m)
  - much larger than taking single tag
- It turns out to be hard to construct efficient MAC with this view (in particular, independent of **Q**)



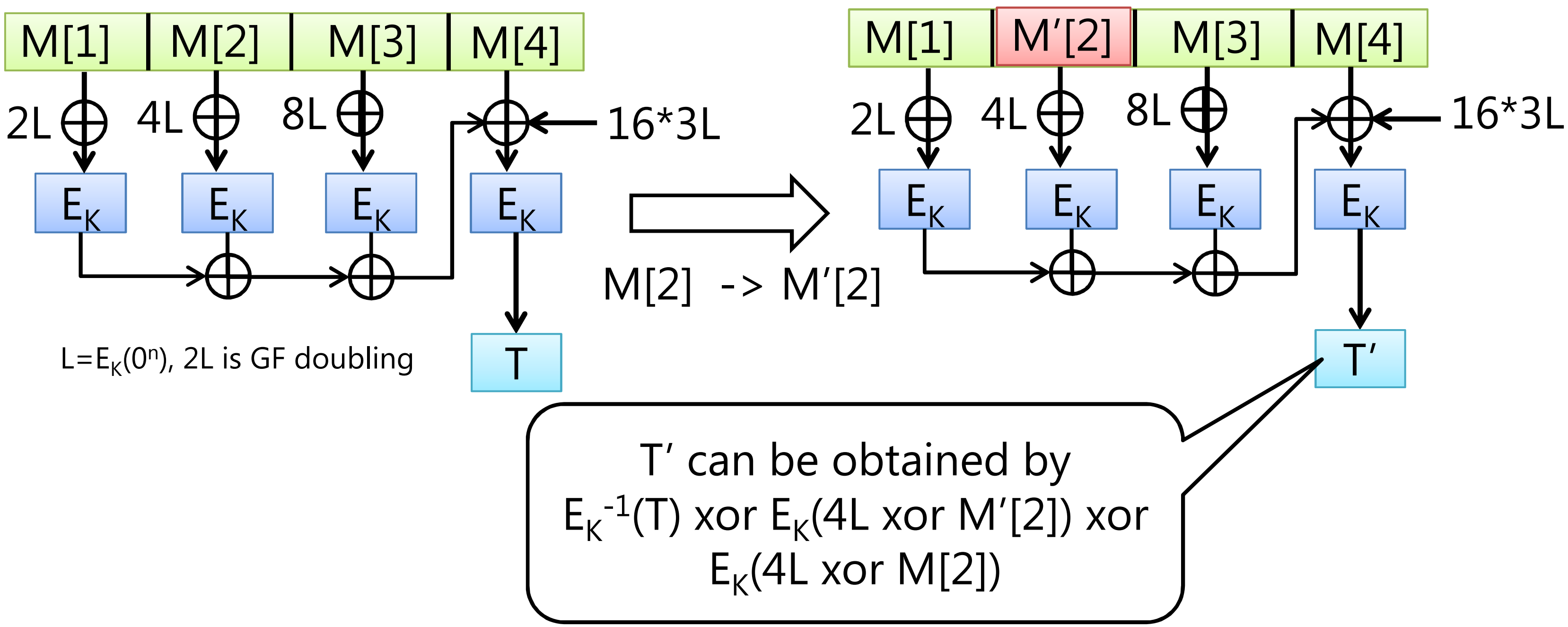
Our view

- Let $\{0,1\}^{\bullet m}$ be the space of extended vectors, where each string can be an *empty string* (ϵ)
- For M $\in \{0,1\}^{*m}$ and B $\in \{0,1\}^{m}$, let M⊙B $\in \{0,1\}^{\bullet m}$ be the extraction with empty string : (M[1],M[2],M[3])⊙(1,0,1) = (M[1], ϵ,M[3])
- Our task is to take T[i] = MAC(M⊙$\mathbf{Q}_i$), where underlying MAC works over $\{0,1\}^{\bullet m}$



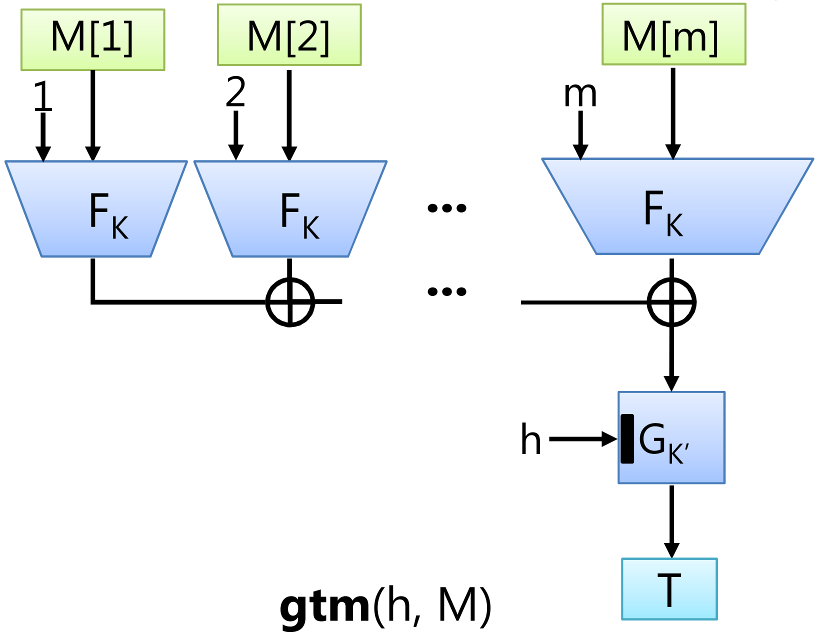
PMAC [Black and Rogaway 02][Rogaway 04]

- A parallelizable, blockcipher-based MAC
  - Defined over string space
  - Each M[i] is non-empty n-bit string (except last one)
  - E_K is an encryption function of n-bit blockcipher (e.g. AES)
- Incremental MAC for "replace" operation
  - Once compute T for M, replace M[i] to M'[i] and recompute T' need few E calls
- Still not suitable for our purpose
  - each block has fixed length, non-empty



Group testing MAC

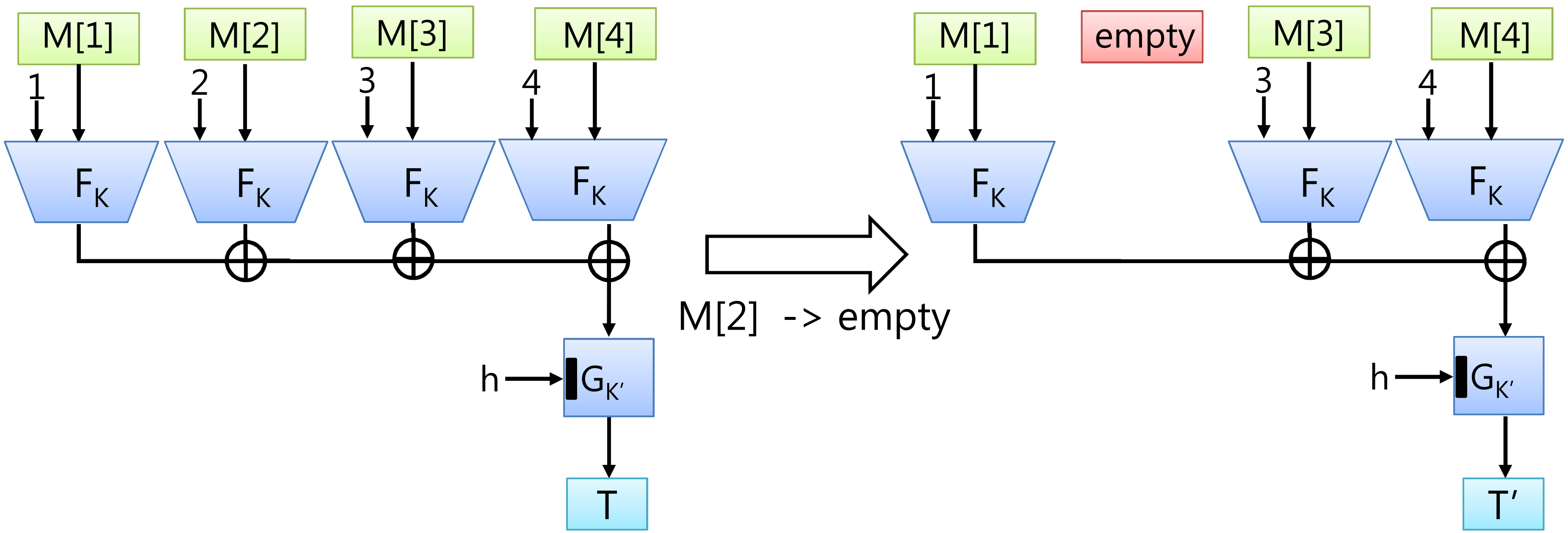
- **gtm** : a generalized & extended PMAC for extended vector space
  - G : n-bit tweakable permutation
  - F : variable-input-length, n-bit output function
    - Two input variables (index, (possibly empty) string)
- G is a tweakable PRP [Liskov-Rivest-Wagner 02]
- F is an *almost* PRF. We require $F(i, \epsilon) = 0^n$ for any i, and otherwise behaves as PRF
  - Can be realized with PRF over non-empty strings



gtm(h, M)

Properties of **gtm**

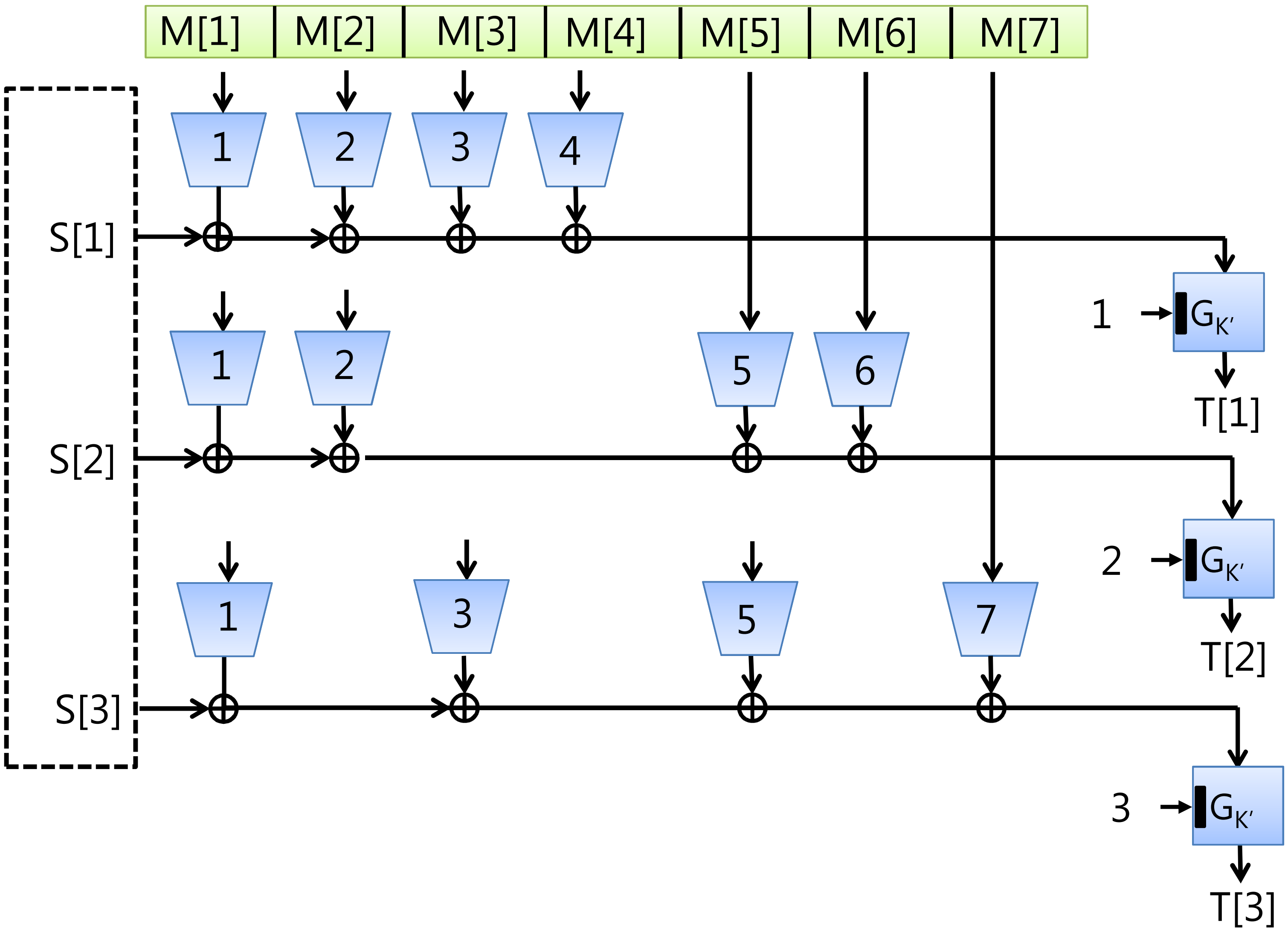
- Provably-secure MAC (PRF) over extended vector space
  - Security proof is mostly the same as PMAC
  - F's fixed point is not a problem (computational XOR-universality is enough, which allows one fixed point)
- We can handle incremental computation, "replace with empty string", in the same manner to PMAC



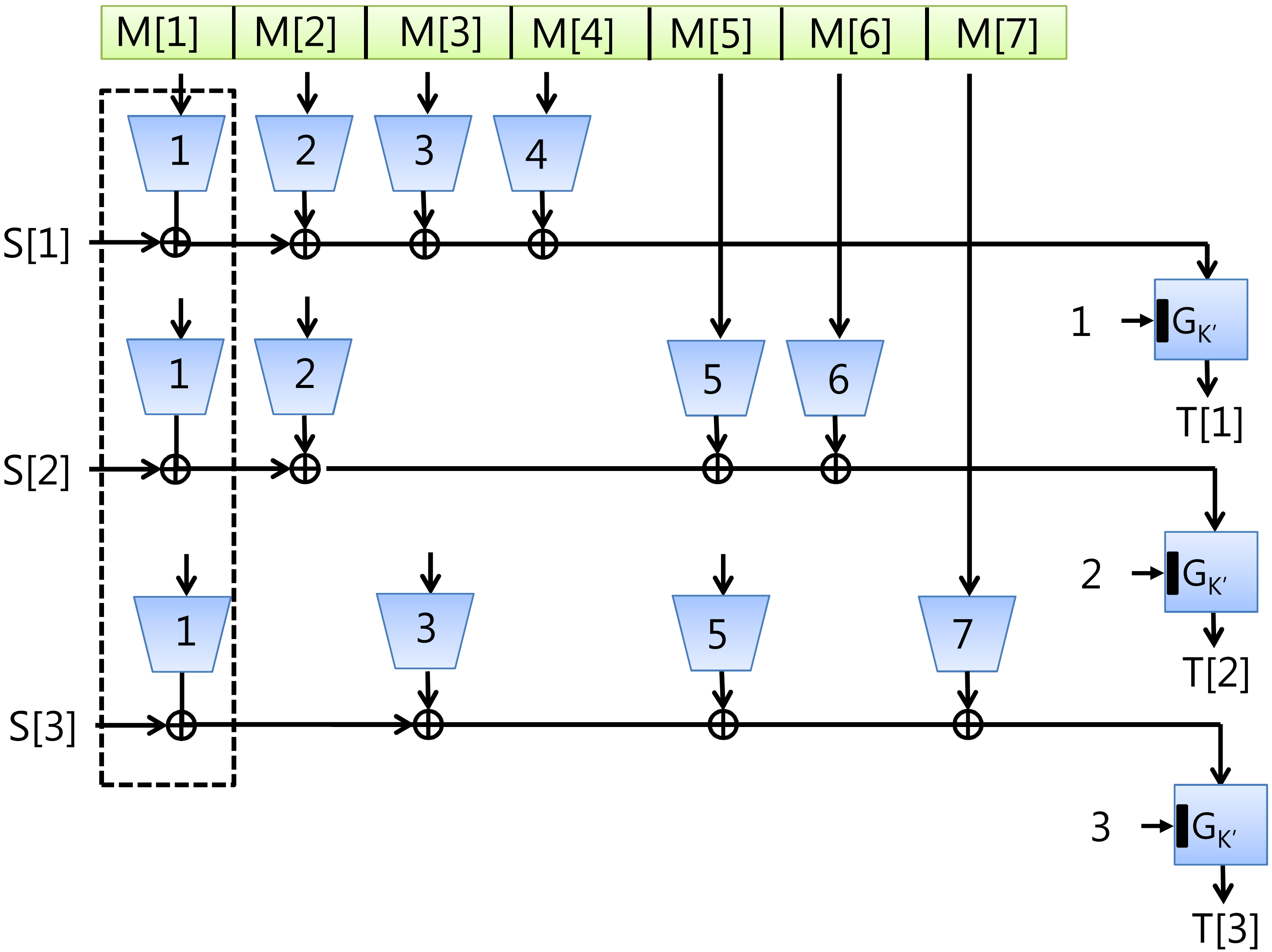
Computing MAC tags with **gtm**

- We compute T[i] = **gtm**(i, M$\odot$$\mathbf{Q}_i$) for i=1,…,t
  - G's tweak (i) is used for security reason
- Ultimately simple method: compute by items
  - Let S[1],…,S[t] be the state variables (initially all-zero)
  - for i = 1,…,m, take Z=F(i,M[i]), add Z to S[j] where $\mathbf{Q}_{i,j}$=1 for all j=1,…,t
  - Output T[j] = G(j,S[j])
- We call this procedure "GTM"

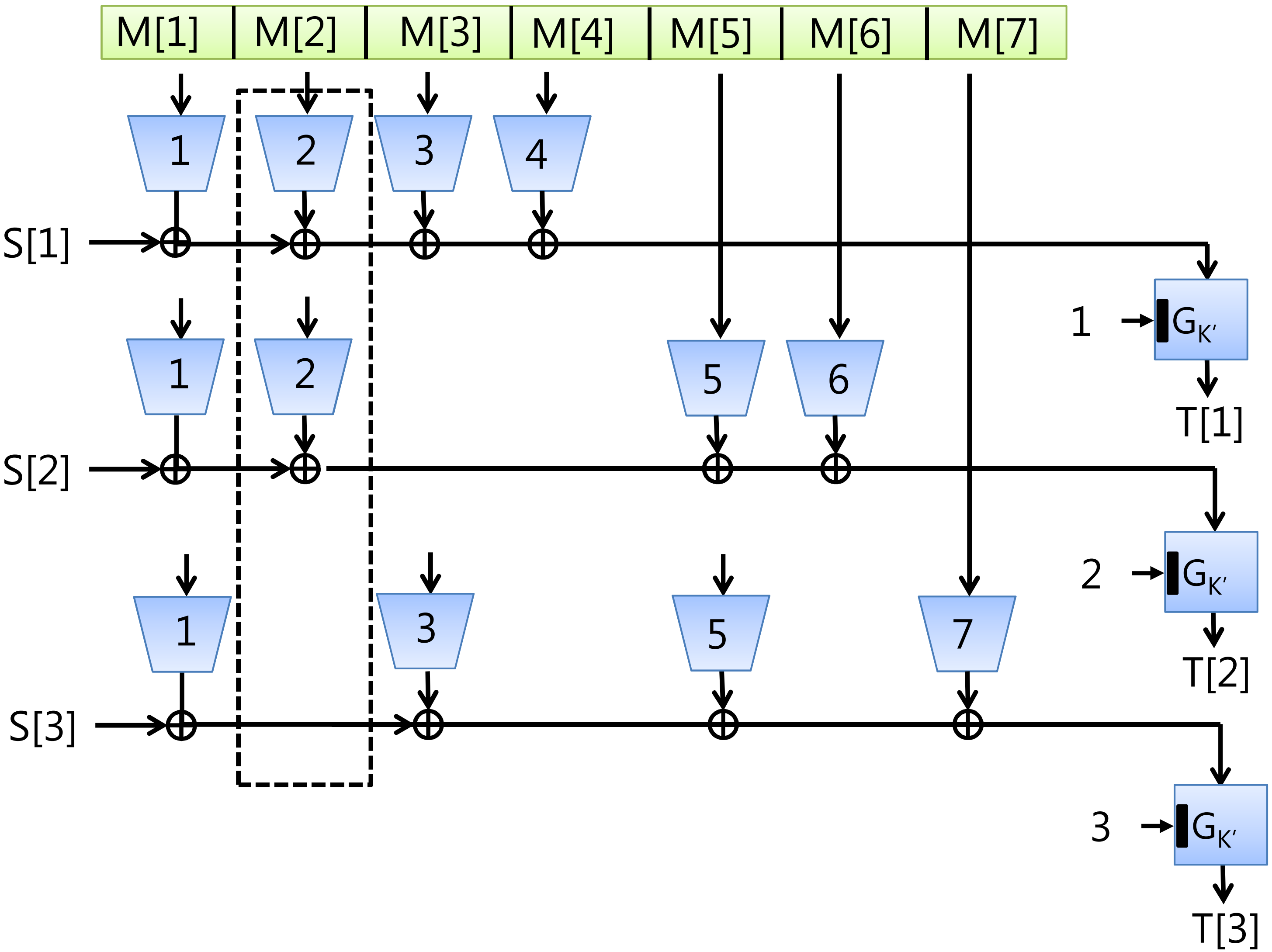
GTM (m=7,t=3,d=1)



GTM (m=7,t=3,d=1)

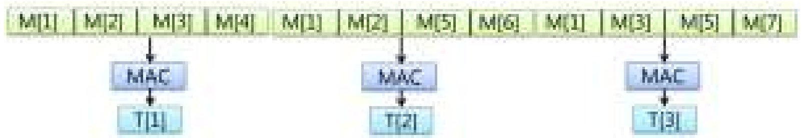


GTM (m=7,t=3,d=1)



Complexity of GTM

- Time : m F calls + t G calls
- Typically, $m \gg t$ and F input $\gg$ G input -> essentially O(m)
- Memory : O(t)
- And this holds for *any* **Q**
  - Can be combined with any known CGT matrix !
- For comparison, naïve method (e.g.) computing $T[i] = \mathbf{gtm}(i, M \ominus \mathbf{Q}_i)$
  - Hw(**Q**) F calls + t G calls -> essentially O(Hw(**Q**)) = O(mt) time, O(t) memory

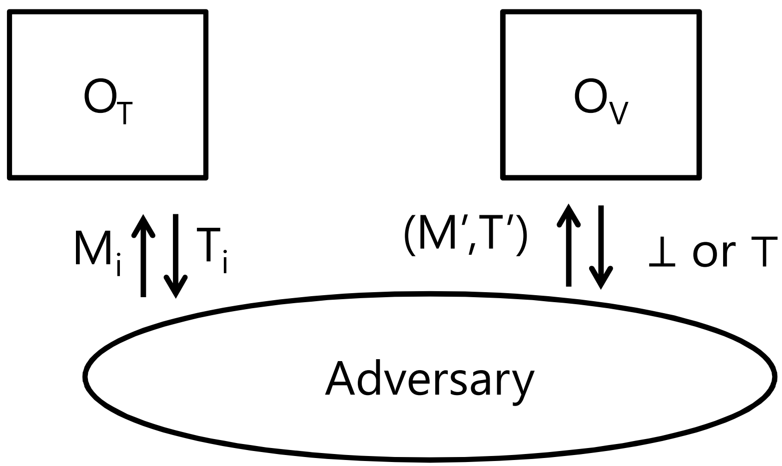


Security

- We considered three notions (for fixed **Q**, t, m)
- Goal : standard deterministic MAC + corruption-finding ability, in a secure manner
- First two notions are about unforgeability
  - Tag vector forgery (TVF) and tag string forgery (TSF) (we omit here)
  - Variants of deterministic MAC security notions
- Third one is about the correctness of corruption identification
  - Corruption misidentification (CM)
  - Hardness of forging naïve decoder's output

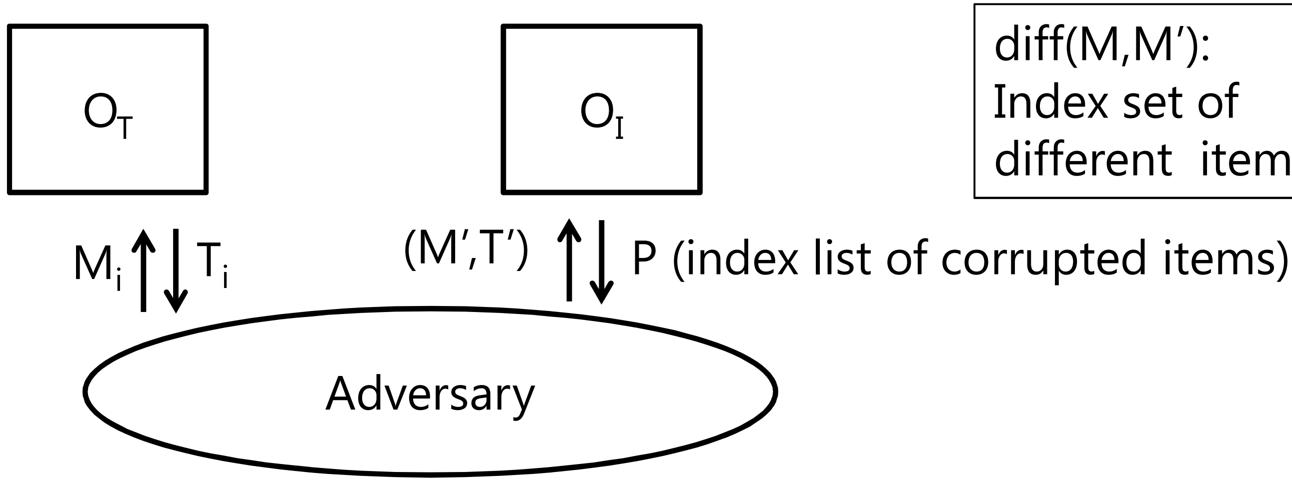
TVF

- Oracles: tagging (O_T) and verification (O_V)
  - O_T takes M and returns T
  - O_V takes (M′,T′) and returns ⊥(invalid) or ⊤(valid)
- adversary *A*
  - first queries O_T and obtains $(M_1,T_1),\ldots,(M_q,T_q)$
  - then queries (M′,T′) to O_V such that
  - $(M',T') \neq (M_i,T_i)$ for all $i=1,\ldots,q$
- *A* wins if O_V's response is valid



CM

- Oracles: tagging (O_T) and identification (O_I) which performs naïve decoding
  - O_T takes M and returns T
  - O_I takes (M',T') and returns {1,…,m}-{i : M[i] is in a negative test}
- d-corruptive adversary *A*
  - first queries O_T and obtains $(M_1,T_1),\ldots,(M_q,T_q)$
  - then queries (M',T') to O_I such that
  - $T' = T_i$ for some i=1,…,q, and $|diff(M',M_i)| \leq d$
- *A* wins if O_I's response is not $diff(M',M_i)$



diff(M,M'):
Index set of
different items

Security analysis

- All notions holds if **gtm** is a secure PRF
- For TVF **Q** needs to contain a standard MAC (i.e. all-one row), otherwise simple attack works
  - **gtm** taking all-one row = MAC for M
  - No performance penalty in practice
- For CM, suppose **Q** is d-disjunct
  - chance to win = a non-trivial collision between tag strings, and w/o non-trivial collision naïve decoder never fails against d-corruptive adversary
- If F and G are ideally secure, and **Q** is d-disjunct and has all-one row, security bounds are $O(q^2t^2/2^n)$ for all three notions

Implementation

CGT methods we use

- We implemented GTM using two CGT methods:
- Shifted traversal design (STD) [Thierry-Mieg 06][Thierry-Mieg-Bailly 08]
  - Composition of simple matrices by rotation and shift
- Chinise Reminder Sieve (CRS) [Eppstein-Goodrich-Hirschberg 07]
  - Number-theoretic construction
- For STD and CRS, matrix generation programs are available
  - Originally, i-th text line = a list of item indexes for T[i]
  - We need to invert it : i-th text line = a list of test indexes using M[i]

Implementation of GTM

- F : CMAC [NIST SP800 38B]
- G : XEX [Rogaway 04]
- Both using AES-128
- Single **gtm** computation for m-block input needs m + few AES calls

- Intel CPU (Ivybridge Core i7 3770 3.4GHz)
  - AES in C runs at 13.3 cycles/byte
- Compared with conventional method (T[i] =**gtm**(i, M$\ominus$$\mathbf{Q}_i$))
- Only implemented tag computation

Results for STD

- Two cases: (m,t) = (940,169) and (2000,121)
- Proposed scheme achieves mostly the same speed as AES for 2Kbyte items
- Speed ratio is quite close to the theory (Hw(**Q**)/m)

Table 1. Implementation results for STD, with parameter (n, q, k).

Parameter $(940, 13, 13)$, $\mathrm{Hw}(\mathbb{Q}) = 12,220$, $\mathrm{Hw}(\mathbb{Q})/m = 13$								
$(m, t) = (940, 169)$	16	32	64	128	256	512	1024	2048
Proposed	63.4	64.0	26.8	20.5	17.3	15.7	14.8	14.4
Conventional	430.2	312.2	249.4	219.8	200.4	190.8	186.7	184.0
Parameter $(2000, 11, 11)$, $\mathrm{Hw}(\mathbb{Q}) = 22,220$, $\mathrm{Hw}(\mathbb{Q})/m = 11.11$								
$(m, t) = (2000, 121)$	16	32	64	128	256	512	1024	2048
Proposed	55.3	33.9	27.3	20.2	16.8	15.1	14.5	14.1
Conventional	361	259.7	206.9	180.7	166.8	159.5	155.9	153.8

Item length (byte)

Speed (cycles/byte)

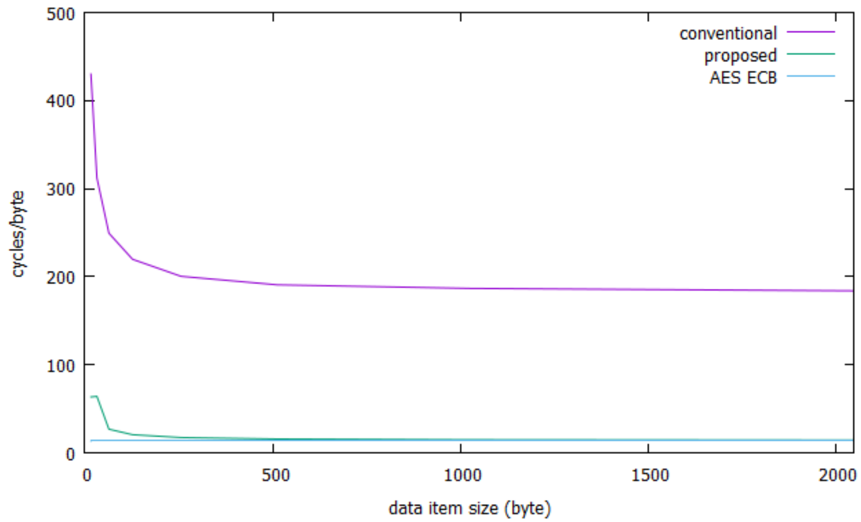
Results for CRS

- Three cases: (m,t)=(10^4,378), (10^4,89) and (10^5,131)
- Similar results as STD
- Improvement factor around 8 ~ 15 (depending on matrix)

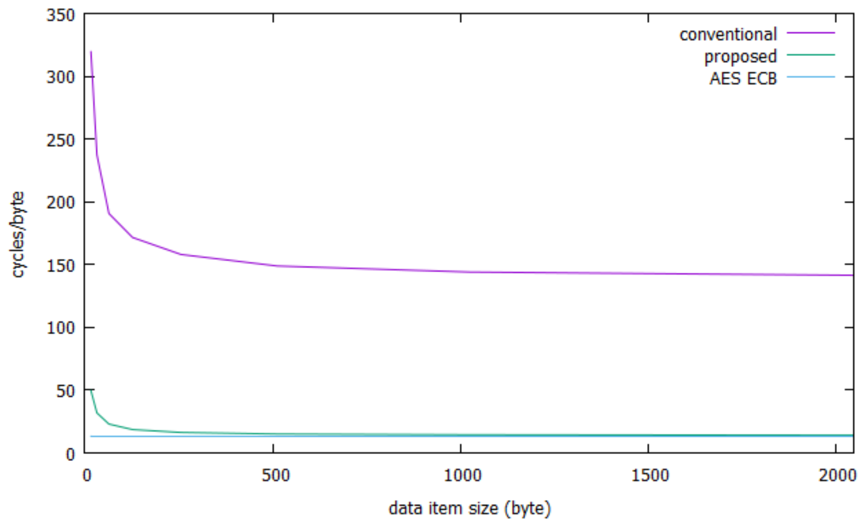
Table 2. Implementation results for CRS, with parameter (n, d).

Parameter $(10^4, 5)$, $\mathrm{Hw}(\mathbb{Q}) = 150,000$, $\mathrm{Hw}(\mathbb{Q})/m = 15$								
$(m,t) = (10^4, 378)$	16	32	64	128	256	512	1024	2048
Proposed	60.9	37.6	25.8	20	17.1	15.6	14.8	14.5
Conventional	492.4	353.5	285	251.4	233	226.9	218.2	215.5
Parameter $(10^4, 2)$, $\mathrm{Hw}(\mathbb{Q}) = 80,000$, $\mathrm{Hw}(\mathbb{Q})/m = 8$								
$(m,t) = (10^4, 89)$	16	32	64	128	256	512	1024	2048
Proposed	51	30.8	22.6	18.4	16.4	15.3	14.7	14.5
Conventional	259.5	189.7	156.1	135.5	125.7	121.2	117.7	116.3
Parameter $(10^5, 2)$, $\mathrm{Hw}(\mathbb{Q}) = 1,000,000$, $\mathrm{Hw}(\mathbb{Q})/m = 10$								
$(m,t) = (10^5, 131)$	16	32	64	128	256	512	1024	2048
Proposed	49.7	31.9	23	18.6	16.3	15.1	14.5	14.1
Conventional	319.6	237.5	190.7	171.6	158.1	148.9	144.1	141.5

Speed comparisons



The case of STD (m,t) = (940,169)



The case of CRS (m,t) = (10^5,131)

Extensions

1. CM-security does not allow the tags to be corrupted
   - When tags are stored separately this is fine, but for communication it is unlikely to hold
2. More relaxed identification
   - Output is a superset of corrupted items with predetermined margin
   - Studied by Corruption-localizing hashing [CJS09]

- Both extensions are possible by using CGT matrix that can tolerate errors at testing
  - Error-correcting list disjunct matrix [Ngo-Porat-Rudra 11] or [Cheraghchi 13]
  - work in progress

Conclusion

- We studied MAC combined with CGT, in particular about its efficiency
- Naively we need O(mt) computations, if we use a CGT matrix of t tests
- Our proposal (GTM) achieves O(m+t) computations (essentially O(m)) for any matrix of t tests
  - using a simple yet non-trivial extension of PMAC
  - proved security in a concrete security framework
- Experimental implementation w/ known CGT matrices demonstrate the effectiveness of our proposal

Thank you!