Updates on Generic Attacks against HMAC and NMAC

Jian Guo[1], Thomas Peyrin[1], Yu Sasaki[2], and Lei Wang[1]

[1] Nanyang Technological University, Singapore
{guojian,thomas.peyrin,wang.lei}@ntu.edu.sg
[2] NTT Secure Platform Laboratories, Japan
sasaki.yu@lab.ntt.co.jp

Abstract. In this paper, we present three different types of new generic attacks against HMAC and other similar MACs when instantiated with an n-bit output hash function maintaining an l-bit internal state. Firstly, we describe two types of selective forgery attacks, in which an adversary commits on the forged message beforehand. The first type is a tight attack which requires $O(2^{l/2})$ computations, while the second one requires $O(2^{2l/3})$ computations, but offers much more freedom degrees in the choice of the committed message. Secondly, we propose an improved universal forgery attack which significantly reduces the complexity of the best known attack from $O(2^{5l/6})$ to $O(2^{3l/4})$. Finally, we describe the very first time-memory tradeoff for key recovery attack on HMAC. With $O(2^{l})$ precomputation, the internal key K_{out} is firstly recovered with $O(2^{2l/3})$ computations by exploiting the Hellman's time-memory tradeoff, and then the other internal key K_{in} is recovered with $O(2^{3l/4})$ computations by a novel approach using a property of functional graph. These results indicate the inefficiency in using long keys for HMAC, *i.e.*, keys with more than l bits does not provide higher security.

Key words: HMAC, NMAC, selective forgery, universal forgery, key recovery

1 Introduction

A message authentication code (MAC) ensures both integrity and authenticity of messages transmitted between two parties sharing a secret key K in advance. When the sender would like to send a message $\mathcal{M}$, he first generates the tag T computed by $T = \mathtt{MAC}(K, \mathcal{M})$, and then sends the pair $(\mathcal{M}, T)$ to the other party. The receiver computes the tag value with the received message using his own key K and checks if this value matches the received tag value T. If they do match, he knows that the message $\mathcal{M}$ received was correct and indeed sent by the other party.

A classical method to build a MAC algorithm is to use a hash function. A well-known example is HMAC [1], designed by Bellare *et al.*, which has been standardized by ANSI, IETF, ISO and NIST, and is widely implemented in various security protocols such as SSL/TLS and IPSec.

There are several security requirements one expects a secure MAC to verify. Informally, it should resist key recovery attacks, any type of forgery attacks, as well as any distinguishing attacks. We note that key recovery and forgeries are arguably the most important security notions as they have the greatest impact in practice. We provide below definitions of the attacks which are related to this paper. It is assumed that the adversary can interact with an oracle that outputs the valid tag $T = \mathtt{MAC}(K, \mathcal{M})$ when queried with a message $\mathcal{M}$.

Key recovery: the adversary recovers the secret-key K used in the MAC algorithm.

Selective forgery: the adversary first commits on a message $\mathcal{M}$ before interacting with the oracle, and then builds a valid pair $(\mathcal{M}, T)$, without having queried $\mathcal{M}$.

Universal forgery: the adversary first receives a message $\mathcal{M}$ sent as challenge, and then builds a valid pair $(\mathcal{M}, T)$, without having queried $\mathcal{M}$.

The security of a MAC construction is discussed in terms of lower-bound and upper-bound on the complexity of the attack for each notion. Regarding the lower-bound, many hash based MACs including HMAC are proven to be indistinguishable from a PRF (pseudo-random function) up to $O(2^{l/2})$ queries, where l is the internal state size of the underlying hash function with n-bit hash digests.

On the other hand, the upper-bound on the complexity is shown by demonstrating a generic attack for each notion. Concerning the notions of existential forgery and the distinguishing-R, Preneel *et al.* proposed a tight generic attack, *i.e.*, the attack complexity matches the proven lower-bound [17]. Their method is based on an internal collision generated with a birthday complexity of $O(2^{l/2})$. Following a similar collision-detection based approach, Naito *et al.* proposed a distinguishing-H attack with a complexity of $O(2^l/l)$ [14]. Although this approach is powerful, its direct application to other notions, particularly the above three defined notions, seems difficult.

Recently, cryptographers have proposed new attack approaches by studying the cycle property of functional graphs and by studying entropy loss of sequential iterations, and applied them to find new generic attacks on hash based MACs [15, 4, 12, 16], as well as dedicated attacks on instances of specific designs [8, 7]. Interestingly, these approaches have even been used to analyze the notions selective forgery and universal forgery. In [16], Peyrin and Wang showed a universal forgery with a complexity of $O(2^{5l/6})$, which is based on cycle property of functional graph. At the same time with our paper, in [3], Dinur and Leurent found another universal forgery with a complexity of $O(2^{6l/7})$ but with shorter queries and thus with wider applications (Peyrin and Wang's attack inherently needs to use queries of $O(2^{l/2})$ blocks long), which is based on collision entropy loss of iterations. We note that selective forgery is a weaker security notion than universal forgery (an attacker can use a universal forgery to generate selective forgeries), and hence these universal forgery attacks can be directly used to obtain a selective forgery with the same complexity.

The current best known universal forgery and selective forgery attacks on hash function based MAC algorithms in [16] are not tight and it remains as an open problem if attacks and/or proofs can be improved. Moreover, as of today, key recovery remains the only security notion for which no generic attack was proposed on hash function based MAC algorithms.

Our contributions. In this article, we present improved selective forgery attacks against HMAC, NMAC, and other similar MACs, as well as an improved universal forgery attack and the very first time-memory tradeoff for key recovery attack.

More precisely, we first describe two types of selective forgery attacks. The first type offers rather limited choice to the adversary regarding the committed message and this message must consist of at least $O(2^{l/2})$ blocks, but the overall complexity is only $O(2^{l/2})$ computations, which matches the proven lower-bound for hash-based MAC algorithms. On the other hand, the second type permits a much broader choice of committed message (the scenario is actually quite close to a universal forgery attack), and its complexity depends on the block length of the committed message. Particularly, the committed message must consist of at least $O(2^{l/3})$ blocks in order to obtain the optimal complexity of $O(l \cdot 2^{2l/3})$ computations. Giving an example with the specifications of widely used hash functions, such as SHA-1 or SHA-256 [20], the adversary can freely choose the committed message, except about 12.5% of it. The former type is a direct application of the distinguishing-H attacks from [12], while the latter is obtained by devising an expandable message technique in the keyed scenario, which was originally proposed by Kelsey and Schneier for keyless hash functions [10]. The obvious main difficulty in the keyed scenario is that the adversary cannot access the internal state values anymore.

Secondly, we improve the complexity of the best known universal forgery attack for hash-based MAC algorithms, which is reduced to $O(\max(2^{l-s}, 2^{3l/4}, 2^s))$ for a challenge message composed of 2^s blocks. Roughly, the complexity has been significantly reduced from $O(2^{5l/6})$ to $O(2^{3l/4})$. Previous universal forgery attacks [16] are based on the analysis of nodes' *height* in the MAC functional graph, the height of a node x being the number of nodes linking x to the cycle of its own component in the functional graph. The basic principle of this attack is that the adversary will first collect offline many values and their exact height in the MAC functional graph, and then use this information to perform the forgery. Unfortunately, the authors failed to estimate the height distributions for the nodes collected offline, which essentially prevents the attack complexity to go below $2^{5l/6}$ computations. In order to overcome this, we performed experiments in order to investigate these height distributions, and we finally observed a very interesting property. This observation remains a conjecture as of today, but confidence in its validity is backed up by our experiments. Based on this conjecture, we managed to improve the universal forgery attack.

Lastly, we propose the first time-memory tradeoff for key recovery attack against HMAC and NMAC. Before discussing our attacks, the key size of HMAC and NMAC needs to be specified. For NMAC instantiated with a l-bit internal state hash function, the key size is l bits for the first key and l bits for the second key, for a total key size of $2l$ bits. HMAC is defined to accept a secret key of an arbitrary size. If the key size is longer than the block size, the key is first hashed by using the underlying hash function, and then the corresponding digest is used as the key. As later explained in Section 2, keys longer than n bits are quite common in industry implementations. Our key recovery attack will target these keys that are larger than n bits. We show that by performing a clever precomputation phase, the second key in NMAC or the equivalent key K_{out} in HMAC can be recovered with a time complexity of $O(2^{2l/3})$ computations and a memory to store $O(2^{2l/3})$ states by applying the Hellman's time-memory tradeoff. After that, the first key in NMAC or the equivalent key K_{in} in HMAC can be recovered with a time complexity of $O(2^{3l/4})$ computations and a memory to store $O(2^{3l/4})$ states.

Paper outline. We recall the HMAC and NMAC specifications in Section 2 and properties of functional graphs in Section 3. Then, we explain the two types of selective forgery attacks in Section 4 and the improved universal forgery attack in Section 5. Finally, we describe the generic key recovery attack in Section 6 and we conclude the paper in Section 7.

2 Description of NMAC and HMAC

A hash function H maps arbitrarily long messages to an n-bit digest. It is usually built by iterating a compression function f, which maps inputs of $l+b$ bits to outputs of l bits. In details, H first pads an input message $\mathcal{M}$ to be a multiple of b bits, then splits it into blocks of b bits each, *i.e.*, $pad(\mathcal{M}) = M_1 \| M_2 \| \cdots \| M_s$, where $\|$ denotes the concatenation operation. It then calls the compression function f iteratively to process these blocks. Finally, H may use a finalization function g that maps l bits to n bits to produce the hash digest. Namely, set $X_0 \leftarrow IV$, compute $X_i \leftarrow f(X_{i-1}, M_i)$ for $i = 1, 2, \ldots, s$, and produce $g(X_s)$ as the final digest, with some finalization function g. Each internal state word X_i is l-bit long, and IV (initial value) is a public constant.

NMAC **algorithm [1]** keys a hash function H by replacing the public IV with a secret key K, which is denoted as H_K. It then uses two l-bit secret keys K_{in} and K_{out} referred to as the inner and the outer keys respectively, and makes two calls to the hash function H. NMAC is simply defined to process an input message $\mathcal{M}$ as NMAC$(K_{out}, K_{in}, \mathcal{M}) = H_{K_{out}}(H_{K_{in}}(\mathcal{M}))$. Keyed functions $H_{K_{in}}$ and $H_{K_{out}}$ are referred to as the inner and the outer (hash) functions respectively.

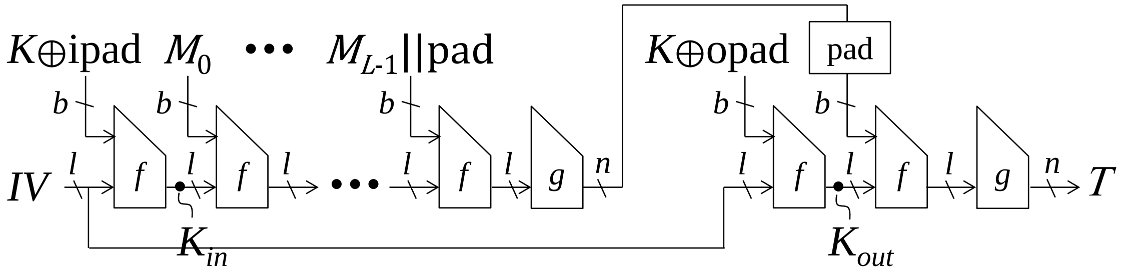

Fig. 1. HMAC with a narrow-pipe hash function

HMAC algorithm [1] is a single-key variant of NMAC, depicted in Figure 1. It derives K_{in} and K_{out} from the single secret key K as $K_{in} = f(IV, K \oplus \texttt{ipad})$ and $K_{out} = f(IV, K \oplus \texttt{opad})$, where ipad and opad are two distinct public constants. HMAC is then simply defined to process an input message $\mathcal{M}$ as $\texttt{HMAC}(K, \mathcal{M}) = H((K \oplus \texttt{opad}) \| H((K \oplus \texttt{ipad}) \| \mathcal{M}))$. HMAC accepts any key size. If the key K is shorter than b bits, then it is padded with 0 bits to reach the size b of a complete compression function message block. Otherwise, if the key K is longer than b bits, then it is hashed and padded with 0 bits: $K \leftarrow H(K) \| 0^{b-n}$.

Regarding the use of keys which are longer than the tag size (n bits), there are both positive and negative decisions by standardization bodies. Indeed, RFC [11] only specifies that n bits is the minimum recommended key size. Though it does not specify the maximum key size, it explains that keys longer than n bits are acceptable, but the extra length would not significantly increase the function strength. However, it recommends longer key sizes when the randomness of the key is considered weak. FIPS [21] specifies that the effective security strength of the HMAC key is the minimum of the security strength of the key and the value of $2l$, where l is the internal state size. Hence, it seems natural to use $2l$-bit keys if that is possible, so as to maximize the security of the construction. Finally, we observe that in fact industry often implements HMAC with much longer key sizes than n bits. This is the case for example in MonoCrypt, which is a cryptographic library currently operated in commerce developed by SBI Net Systems [19]. MonoCrypt supports 80-bit, 128-bit, 512-bit, 576-bits, and 640-bit keys for HMAC-SHA-1 and 160-bit, 192-bit, 512-bit, 576-bit, and 640-bit keys for HMAC-SHA-256.

For simplicity, hereafter we will describe the attacks based on HMAC. However, we emphasize that our methods apply similarly to hash function based MACs such as NMAC [1] and Sandwich-MAC [22].

3 Functional graph

In this article and in previous works on HMAC cryptanalysis [15, 12, 16], the analysis of properties of functional graphs for random functions is very important. We recall a few results in this section.

The functional graph $\mathcal{G}_f$ of a function $f : \{0,1\}^l \to \{0,1\}^l$ is simply the directed graph in which the vertices (or nodes) are all the values in $\{0,1\}^l$ and where the directed edges are the iterations of f (i.e. a directed edge from a vertex a to a vertex b exists if and only if $f(a) = b$). The functional graph of a function is composed of one or several components, each having its own internal cycle.

The following Theorems 1, 2 and 3 state several statistical properties of the functional graph of a random function.

Theorem 1 ([5, Th. 2]). *The expectations of the number of components, number of cyclic nodes (a node belonging to the cycle of its component), number of terminal nodes (a node without a preimage), and number of image nodes (a node with a preimage) in a random mapping of size N have the asymptotic forms, as $N \to \infty$:*

(i) #Components: $\frac{1}{2}\log N$

(ii) #Cyclic nodes: $\sqrt{\pi N/2}$

(iii) #Terminal nodes: $e^{-1}N$

(iv) #Image nodes: $(1 - e^{-1})N$

Starting from any node x, the iteration structure of f is described by a simple path that connects to a cycle. The length of the path (measured by the number of edges) is called the tail length of x (or the height of x) and is denoted by $\lambda(x)$. The length of the cycle is called the cycle length of x and is denoted $\mu(x)$. Finally, the rho-length of x is denoted $\rho(x)$ and represents the length of the non repeating trajectory of x: $\rho(x) = \lambda(x) + \mu(x)$.

Theorem 2 ([5, Th. 3]). *Seen from a random node in a random mapping of size N, the expectations of the tail length, cycle length, rho length, tree size, component size, and predecessors size have the following asymptotic forms:*

(i) Tail length (λ)*:* $\sqrt{\pi N/8}$
(ii) Cycle length (μ)*:* $\sqrt{\pi N/8}$
(iii) Rho length ($\rho = \lambda + \mu$)*:* $\sqrt{\pi N/2}$
(iv) Tree size: $N/3$
(v) Component size: $2N/3$
(vi) Predecessors size: $\sqrt{\pi N/8}$

Moreover, the asymptotic expectations of the giant component and its giant tree have been provided in [6].

Theorem 3 ([6, VII.14]). *In a random mapping of size* N*, the largest tree and the largest component have expectations asymptotic, respectively, of* $0.48 * N$ *and* $0.7582 * N$*.*

In this article, we will study the functional graph of a compression function f, when a constant value M is used as message block input (*i.e.* the function f is iterated with fixed messages block all equal to M). We will denote such a function as $f_M : \{0,1\}^l \rightarrow \{0,1\}^l$, and $\mathcal{G}_{f_M}$ its corresponding functional graph.

4 Selective forgery attacks

In this section, we show two types of generic selective forgery attacks against HMAC. The attacker first commits on some message $\mathcal{M}$, and then can interact with the MAC oracle to output the valid tag T corresponding to $\mathcal{M}$ without querying $\mathcal{M}$. Note that the offline phase refers to the computations done before committing on $\mathcal{M}$, while the online phase refers to the computations done and the queries sent after committing on $\mathcal{M}$. Moreover, we denote $M^{(i)}$ the i successive concatenation of M.

4.1 Attack with a very constrained target message

The adversary will have to choose a long message, composed of $O(2^{l/2})$ blocks. Our method is a direct application of the distinguishing-H technique by [12].

Committing Phase (Offline)

1. As done in [12], draw the functional graph $\mathcal{G}_{f_M}$ of the underlying compression function f where a fixed message block M is used as the input message block, and compute the size γ of the cycle of the largest component.

2. Select as target message for the selective forgery the message $\mathcal{M} = M^{(2^{l/2})} \| M' \| M^{(2^{l/2+\gamma})}$, where M' can be any message block such that $M' \neq M$.

Challenging Phase (Online)

3. Query $M^{(2^{l/2+\gamma})} \| M' \| M^{(2^{l/2})}$ to obtain the tag T. Output the pair $(\mathcal{M}, T)$.

The complexity and success probability evaluation is exactly the same as in [12] and we refer to the original article for more details. Informally, since a least $2^{l/2}$ identical message blocks M are used as prefix and suffix, there is a good chance that we enter in the main cycle of the functional graph $\mathcal{G}_{f_M}$ before and after processing message block M'. If this is true for $\mathcal{M}$ then it will be true for the queried message $M^{(2^{l/2+\gamma})} \| M' \| M^{(2^{l/2})}$ as well, and both will be fully synchronized inside the cycle, which means that they will end up to the same tag value. The overall success probability is equal to 0.14, but if needed it can be improved by iterating the fixed message block M a little bit more. The overall attack complexity is $O(2^{l/2})$ computations, which matches the proven lower-bound of the HMAC construction. This attack is therefore tight and it closes the discussions on the security gap of HMAC with regards to selective forgery notion.

Concerning the choice of the target message $\mathcal{M}$ by the adversary, we note that he can freely choose the values of M and M', but he can also append any prefix and suffix while preserving the validity of the attack. However, the target message $\mathcal{M}$ eventually contains quite a long iteration of an identical message block M, which constrains a lot the adversary's freedom to choose it.

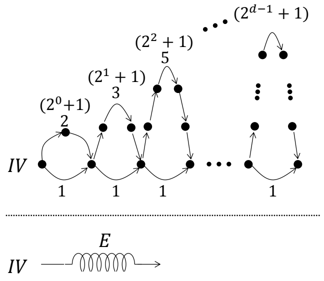

Fig. 2. (Top) Expandable message ranging from d to $d+2^d-1$ blocks. (Bottom) Simplified representation

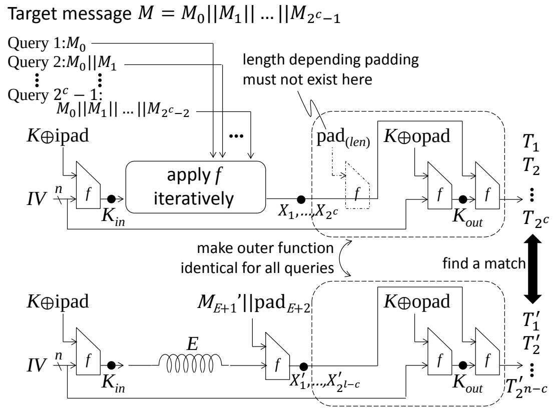
Target message $M = M_0||M_1||\ldots||M_{2^c-1}$
Query 1:M_0
Query 2:$M_0||M_1$
Query 2^c-1: $M_0||M_1||\ldots||M_{2^c-2}$
length depending padding must not exist here
$K\oplus$ipad
pad$_{(len)}$
$K\oplus$opad
IV
n
f
apply f iteratively
K_{in}
$X_1,\ldots,X_{2^c}$
K_{out}
T_1
T_2
T_{2^c}
make outer function identical for all queries
find a match
$K\oplus$ipad
$M_{E+1}'||\text{pad}_{E+2}$
$K\oplus$opad
E
$X'_1,\ldots,X'_{2^{l-c}}$
T'_1
T'_2
$T'_{2^{n-c}}$

Fig. 3. Strategy for computing a selective forgery with more freedom degrees on the target message.

4.2 Attack with more freedom degrees on the target message

The padding scheme used in the underlying hash function is heavily related to this analysis. Here, we suppose the MD-strengthening padding, which is widely used in practice e.g. by `SHA-1` or `SHA-256` [20]. Its essence is appending the message length information to the end of the message. See [20] for details.

In Section 4.2, we suppose that the underlying hash function is narrow-pipe, i.e. $l \leftarrow n$.

Our selective forgery attack uses a strategy similar to the previous generic second-preimage attack for hash functions [2, 10], which can generate a second-preimage with a complexity of 2^{n-c} for a target message of size 2^c blocks. We briefly recall previous second-preimage attacks on hash functions. The initial idea is to try 2^{n-c} random messages in order to find one that collides with one value of the 2^c internal chaining variables of the target message. However, the pre-specified message length in the padding string prevents this naive attack. Kelsey and Schneier showed that this issue can be solved with a multi-collision consisting of messages with different block lengths [10]. The generated multi-collision structure is called an *expandable message*. Informally, it generates a collision between a 1-block message and a $1 + 2^0 = 2$-block message, followed by a collision between an 1-block message and a $1 + 2^1 = 3$-block message. Similarly, a collision between an 1-block message and a $(1+2^i)$-block message is generated for $i = 0, 1, \ldots, d-1$. Then, any block length from d to $d + 2^d - 1$ can be reached by choosing the appropriate combination of the message blocks. An example is shown in Figure 2.

Adapting the generic second-preimage attacks for hash functions to compute selective forgery in the setting of `MAC`'s presents several main difficulties:

1. Due to the two equivalent secret keys in `HMAC`, the adversary cannot access the internal state values after each message block is processed.
2. If the length of an input message changes, the MD-strengthening padding for the inner function will be different and thus the result of the outer function will change as well. This makes it difficult to cut an input message and analyze only up to the exact middle (i-th) message block.

Attack overview. Before going into details, we explain our strategy, which is illustrated in Figure 3. To begin, we solve the first issue. In our attack, instead of storing the internal state value after each message block (which is unknown because of the secret key used in the `MAC`), the adversary queries the first i blocks of the 2^c-block target message for $i = 1, 2, \ldots, 2^c-1$, and stores the corresponding tags. Namely, the first query is M_0, the second query is $M_0 \| M_1$, the third query is $M_0 \| M_1 \| M_2$, and so on. Let X_i be the unknown internal state value after processing the i-th message block and T_i its corresponding tag value. This is illustrated in top of Figure 3. The adversary later searches for a connection from the target message to the 2^{n-c}-block second message. Let X'_j be the output of the inner function (the internal state) and T'_j its corresponding tag value for the second message, where $j = 1, 2, \ldots, 2^{n-c}$. This is illustrated in bottom of Figure 3. If we can make the function from X_i to T_i and the function from X'_j to T'_j identical, a collision on the tag (*i.e.* $T_i = T'_j$) suggests a collision of the internal state (*i.e.* $X_i = X'_j$) with a good probability. Therefore, a connection from the second message to the target message can be found just by looking at the tag outputs, without even knowing the internal state values.

We then explain our strategy to solve the second issue. As indicated in Figure 3, the function from X_i to T_i may process the padding block, which depends on the input message length. This padding block issue can be avoided by selecting the target message so that the padding string is embedded inside each message block. Namely, message block M_i is chosen to be composed of $M_i = m_i \| p_i$ where m_i can be any value, and where p_i is the padding string corresponding to a message of $i+1$ blocks plus the bit length of $|m_i|$.

With these few tricks, we can adapt the second-preimage attacks to the `MAC` setting and the selective forgery attack can eventually be carried out successfully.

Attack procedure. Our attack is divided into five steps: 1) selecting the target message, 2) obtaining T_i, 3) building an expandable message, 4) obtaining T'_j, and 5) forging the tag. Only the first step is done offline, the rest being online.

1. **Selecting the target message.** The attacker must commit on the target message $\mathcal{M}$ of length $c+2^c+1$ blocks[3], where c is a parameter for the attack that we will determine later. More precisely, he will choose $\mathcal{M} = M_{-c} \| \cdots \| M_{-1} \| M_0 \| \cdots \| M_{2^c-1} \| M_{2^c}$, where the first c blocks $M_{-c} \| \cdots \| M_{-1}$ and the last block M_{2^c} can take any value of his choice. For the middle 2^c blocks from M_0 to M_{2^c-1}, he will set each block as $M_i \leftarrow m_i \| p_i$, where m_i can be set to any value of his choice and where p_i is the padding string corresponding to $1+c+i$ blocks plus $|m_i|$ bits (here the "$1+c$" corresponds to the first $1+c$ blocks $(K \oplus \texttt{ipad}) \| M_{-c} \| \cdots \| M_{-1}$ that will be handled by the internal hash function).
2. **Obtaining T_i.** After committing on $\mathcal{M}$, the online part can start:
   1. Query $M_{-c} \| \cdots \| M_{-1} \| m_0$ and store the tag T_1 in a list L.
   2. Query $M_{-c} \| \cdots \| M_{-1} \| M_0 \| m_1$ and store the tag T_2 in L.

   i. Query $M_{-c} \| \cdots \| M_{-1} \| M_0 \| \cdots \| M_{i-2} \| m_{i-1}$ and store the tag T_i in L for $i = 3, 4, \ldots, 2^c$.

   For the query at step i, due to the padding process the last message block becomes $m_{i-1} \| p_{i-1}$ which is in fact M_{i-1}.
3. **Building an expandable message.** The adversary builds an expandable message ranging from c blocks to $c + 2^c - 1$ blocks in order to later have the possibility to freely adjust the length. E_{prev} will denote the shortest colliding message discovered so far and is naturally initialized to a `null` string. Then, the following procedure is iterated for $i = 0, 1, \ldots, c-1$:

   3.1 Choose $2^{n/2}$ distinct 1-block messages $E_i[u]$. Query $E_{prev} \| E_i[u]$ and store the tags in a list L_u.

   3.2 Choose $2^{n/2}$ distinct (2^i+1)-block messages $E'_i[v]$. Query $E_{prev} \| E'_i[v]$ and store the tags in a list L_v.

   3.3 Find a match between L_u and L_v. Let $\hat{u}$ and $\hat{v}$ be the matched indices.

   3.4 To eliminate the false positives, find a collision by appending $2^{n/2}$ distinct single block messages after $E_{prev} \| E_i[\hat{u}] \| p_{\hat{u}}$, where $p_{\hat{u}}$ is the corresponding padding bits. Let E_x and E'_x be two messages that lead to a collision.

   3.5 Query $E_{prev} \| E'_i[\hat{v}] \| p_{\hat{v}} \| E_x$ and $E_{prev} \| E'_i[\hat{v}] \| p_{\hat{v}} \| E'_x$. If their tags collide, $E_i[\hat{u}]$ and $E_i[\hat{v}]$ are internal collisions. Store $E_i[\hat{u}] \| p_{\hat{u}}$ and $E'_i[\hat{v}] \| p_{\hat{v}}$ as the i-th colliding pair of the expandable message. Otherwise, they are false positive, and we continue the search.

   3.6 Update $E_{prev} \leftarrow E_{prev} \| E_i[\hat{u}] \| p_{\hat{u}}$.

[3] "c" comes from the minimum length of the expandable message and "+1" comes from the last message block. The detailed reasoning for the "+1" is explained later.

The number of queries for Step 3.1 and Step 3.2 for the i-th block is $(i+1)\cdot 2^{n/2}$ and $(i+1+2^i)\cdot 2^{n/2}$ respectively, which is unbalanced. For optimization, we generate more shorter messages $E_{prev}\|E_i[u]$ than longer messages $E_{prev}\|E'_i[v]$. Let α and β be $i+1$ and 2^i+i+1, respectively. We get balance between the two query costs by generating $2^{n/2+(\log\beta-\log\alpha)/2}$ choices of $E_{prev}\|E_i[u]$ and $2^{n/2-(\log\beta-\log\alpha)/2}$ choices of $E_{prev}\|E'_i[v]$. The entire cost is the sum of two costs over the c iterations, $\sum_{i=0}^{c-1} 2^{(n/2+\log\beta+\log\alpha)/2+1}$, which amounts to $O(c\cdot 2^{n/2+c/2})$ blocks of queries. The memory cost is for storing tag values for $E_{prev}\|E'_i[v]$ in which the number of generation is smaller than the tag values for $E_{prev}\|E_i[u]$. Hence, the memory cost is $2^{n/2-(\log(2^i+i+1)-\log(i+1))/2}$ When $i\leftarrow c$, the memory cost is $O(2^{n/2-c/2})$. The cost for eliminating false positives at Step 3.4 is $(i+1)\cdot 2^{n/2}$, which is smaller than Steps 3.1 and 3.2.

4. **Obtaining T'_j.** The length of the expandable message is at minimum c blocks, and we let M'_E denote this shortest c-block instance of the expandable message. Then, $(K_{in}\oplus \texttt{ipad})\|M'_E$ fits in $1+c$ blocks. The adversary generates 2^j distinct 1-block message $M'_j = m'_j\|p'$ for $j=1,2,\ldots,2^{n-c}$, where p' is the padding string for messages of $1+c$ blocks plus $|m'_j|$ bits long. Query $M'_E\|m'_j$ for $j=1,2,\ldots,2^{n-c}$, and store the received tag T'_j in a list L'.
5. **Forging the tag.** Because 2^c T_i values are stored in L and 2^{n-c} T'_j values are stored in L', we expect to find a match between T_i and T'_j. With a good probability, the corresponding X_i and X'_j are also colliding. Then, the length of the expandable message is adjusted to be equal to $c+i-1$ blocks so that the length of the expandable message followed by the block M'_j can be the same as the length of $M_{-c}\|\cdots\|M_{-1}\|M_0\|\cdots\|M_{i-1}$. We denote $\overline{M'}$ the message chunk build by concatenating the length-adjusted expandable message and M'_j.

   These two messages have the same length and result in the same internal state value. Thus, the adversary can append $M_i\|M_{i+1}\|\cdots\|M_{2^c}$ to the end of $\overline{M'}$, and query to the oracle this newly formed message. The received tag value $\overline{T}$ is also a valid tag for the selected target message $\mathcal{M}$.

   Note that we need to ensure that the match is done before the last message block of $\mathcal{M}$, so that we have at least 1 block appended to $\overline{M'}$. That is the reason why we add the last block M_{2^c}.

Complexity evaluation. We proceed the complexity evaluation of our attack. First, one can see that Step 1 is negligible, while Step 2 requires to query $c{+}1, c{+}2, \ldots, c{+}2^c$ blocks, which amounts $c{\cdot}2^c{+}2^c{\cdot}(2^c{+}1)/2 \approx 2^{2c}$ blocks in total. It also requires a memory sufficient to store 2^c tags. In Step 3, the cost is $O(c\cdot 2^{n/2+c/2})$ queries and $O(2^{n/2-c/2})$ tags as explained previously, and the memory cost is equivalent to $O(2^{n/2-c/2})$. tags. Step 4 requires to query $(c+1)\cdot 2^{n-c}$ blocks and a memory to store 2^{n-c} tags, while Step 5 is negligible (one can combine Step 4 and Step 5 so that values generated at Step 4 are tested immediately, which would render this part memoryless).

In total, the number of queries is about $2^{2c} + c\cdot 2^{n/2+c/2} + (c+1)\cdot 2^{n-c}$ blocks, which is minimized to $O(n\cdot 2^{2n/3})$ blocks when $c = n/3$. The memory requirement is $O(2^c + 2^{n/2-c/2})$, which would become $O(2^{n/3})$ when $c = n/3$.

5 Improved universal forgery attacks

In this section, we show an improved generic universal forgery attack against HMAC. We recall that for universal forgeries, the attacker is first challenged with a message $\mathcal{M}$, and after interacting with the MAC oracle he must output the valid tag T corresponding to $\mathcal{M}$ (without querying $\mathcal{M}$ to the oracle).

5.1 Revisiting previous universal forgery attacks on HMAC and NMAC

Recently, Peyrin and Wang published a universal forgery attack on iterative hash function-based MACs [16]. Their attack use a special property: the height of a node in a functional graph. We recall that in a functional graph each node x has a unique path connecting it with a cycle node, and the length of this path is called the *height* of x and is denoted as $\lambda(x)$. A brief description of their attack is provided below.

Let $\mathcal{M} = M_1 \| M_2 \| \cdots \| M_{2^s}$ be the given challenge message (after padding) for the universal forgery, and $X = \{X_1, X_2, \ldots, X_{2^s}\}$ be the successive internal state values during the processing of $\mathcal{M}$ in inner hash function, where X_i denotes the internal state after $M_1 \| \cdots \| M_i$ has been processed.

The attacker computes the height in a functional graph of 2^{s_1} ($s_1 < s$) unknown internal state values during the processing of the challenge message. Meanwhile he also collects 2^{l-s_1} offline values with their heights in the same functional graph. Note there is a good probability that one unknown internal state value collides with one offline collected value, which of course have the same height value. Then the attacker deduces the exact value of one of the unknown internal state values by identifying such a collision pair. In details, the attacker first matches the height values between the unknown internal state values and the offline collected values, which exponentially reduce the candidate pairs, and then examines each remaining pair individually. Finally, Once an internal state value is recovered, a classical second-preimage-like attack trivially allows to compute a universal forgery for the challenge.

However, Peyrin and Wang left an open problem, that is the height distribution in the set of the offline collected values. It is essential in order to obtain *tighter* upper bound of the attack complexity, and thus deserves further investigation. Due to the limited space, we refer to [16] for detailed argument. Here we mainly recall the procedure of collecting offline values and computing their heights, and then illustrate why it is hard to analyze their height distribution. Let $\mathcal{G}_{f_M}$ be the functional graph used in the attack, where V is a random message block value chosen by the attacker. The procedure is described below.

1. Initialize a table Y to be empty.
2. Select a random node y_1 such that $y_1 \notin Y$.
3. Iteratively compute $y_i = f_V(y_{i-1})$ until either of two cases occur:
   - y_i collides with a previously stored nodes in Y; or
   - y_i collides with a previous node y_j $(1 \leq j \leq i-1)$ in the currently computed chain, namely a new cycle is generated.
4. Compute the height values for all nodes $y_1, \ldots, y_i$ in the chain, and store them in Y.
5. Repeat Steps $2-4$ until the number of nodes in Y becomes 2^{l-s_1}.

As we see, it is quite a difficult task to analyze the height distribution in the set Y because the nodes are not chosen uniformly (the process does not pick each node individually and randomly, but it picks a node y_1 and then picks all the nodes in the chain from y_1 to the cycle of y_1's component in the functional graph $\mathcal{G}_{f_M}$).

5.2 Our observations

We have experimentally investigated the height distributions in the set Y, which is generated by the procedure in Section 5.1. Denote by Y_λ a subset of nodes in Y that have the height value λ, and by $|Y_\lambda|$ the number of nodes in Y_λ. In our experiment, we mainly pay attention on finding the smallest height value λ such that $|Y_\lambda|$ is *asymptotically* less than $2^{l/2-s}$, and observed an interesting phenomenon. Although we did not manage to prove formally this observation, we state this reasonable conjecture below.

Conjecture 1. If in total 2^t distinct nodes, where $l/2 \leq t \leq l$ holds, are collected following the procedure in Section 5.1, then for any integer λ satisfying $1 \leq \lambda \leq 2^{l/2}/l$, there are $\Theta(2^{t-l/2})$ nodes collected with the height value λ.

The extreme cases $t = l/2$ and $t = l$ are easy to analyze. For the case $t = l/2$, a randomly selected starting node has a height value $\Theta(2^{l/2})$ on average, and then for each height value λ such that $1 \leq \lambda \leq 2^{l/2}/l$ holds, $\Theta(1)$ nodes will be collected. For the case $t = l$, researchers have already carried out extensive studies on this topic. The set of all nodes with the same height λ is usually called the λ-th *stratum* of the functional graph, and we denote it as S_λ. Particularly, Mutafchiev [13] has proven the following theorem.

Theorem 4 ([13, Lemma 2]). *If $l \to \infty$ and $\lambda = o(2^{l/2})$, the mean value of the λ-th stratum S_λ is $\sqrt{\pi/2} * 2^{l/2}$.*

Since $2^{l/2}/l = o(2^{l/2})$ indeed holds, we get that Conjecture 1 has actually already been proven for the case $t = l$.

To further verify Conjecture 1, we performed experiments for small values of l (namely we computed the smallest value of the subset size $|Y_i|$ for $1 \leq i \leq 2^{l/2}/l$), which will be reported in full version of the paper.

5.3 Improved universal forgery attacks

We present an improved universal forgery attack based on Conjecture 1. We divide $\mathcal{M}$ into two parts: $M_1 \| \cdots \| M_{2^{s_1}}$ and $M_{2^{s_1}+1} \| \cdots \| M_{2^s}$ with $s_1 \leq s - 1$.

1. (online) Recover the height value $\lambda(X_i)$ of each X_i with $1 \leq i \leq 2^{s_1}$ in the functional graph $\mathcal{G}_{f_M}$. For the interested reader, the exact procedure of this step is referred to [16]. For each X_i the complexity of evaluating its height is $O(2^{l/2})$.

2. (online) Find a pair of 1-block message (m, m') with a birthday-like collision attack, such that $M_1 \| \cdots \| M_{2^{s_1}} \| m$ and $M_1 \| \cdots \| M_{2^{s_1}} \| m'$ is a collision on the inner hash function. The complexity is upper-bounded by $O(2^{s_1+l/2})$.
Moreover, it is important to notice that (m, m') is a filter for all X_i with $1 \leq i \leq 2^{s_1}$ as the relation below holds:

$$f(f(\cdots f(X_i, M_{i+1}) \cdots, M_{2^{s_1}}), m) = f(f(\cdots f(X_i, M_{i+1}) \cdots, M_{2^{s_1}}), m').$$

3. (offline) Use the same collection procedure with previous attacks [16] to select 2^{l-s_1} nodes and obtain their respective height in the functional graph $\mathcal{G}_{f_M}$. However, in contrary to the previous attack, we only store the nodes with height λ satisfying $0 \leq \lambda \leq 2^{l/2}/l$. Moreover, for each such height λ, we store exactly $2^{l/2-s_1}$ nodes in Y in the end. According to Conjecture 1, we just need to repeat the collection procedure by at most a constant number of times. Thus, the complexity of this step is upper-bounded by $O(2^{l-s_1})$. It is important to recall that we now know the height distribution for the selected nodes in the set Y. More precisely, for each height λ such that $0 \leq \lambda \leq 2^{l/2}/l$ hold, there are $2^{l/2-s_1}$ nodes in Y that have height λ.

4. (offline) Recover the value of some X_i by matching the elements between X and Y. In details, for each X_i, if $\lambda(X_i) \leq 2^{l/2}/l$ holds, then:

4.1 Obtain the elements in Y that have the height value $\lambda(X_i)$. Let them be a subset of Y denoted as $Y_{\lambda(X_i)}$. We know that $|Y_{\lambda(X_i)}| = 2^{l/2-s_1}$ holds.

4.2 For each node y in Y with height $\lambda(X_i)$, check if the following holds

$$f(f(\cdots f(y, M_{i+1}) \cdots, M_{2^{s_1}}), m) = f(f(\cdots f(y, M_{i+1}) \cdots, M_{2^{s_1}}), m').$$

and if it does, then output the value of y as the value of X_i.

The complexity of this step for a single X_i is computed as $(2^{s_1} - i) \cdot |Y_{\lambda(X_i)}| = (2^{s_1} - i) \cdot 2^{l/2-s_1} = 2^{l/2} - i \cdot 2^{l/2-s_1}$ and so the total complexity of this step is given by

$$\sum_{i=1}^{2^{s_1}} (2^{l/2} - i \cdot 2^{l/2-s_1}) = O(2^{s_1+l/2})$$

5. (offline) Based on the knowledge of some intermediate hash value X_i, construct a second-preimage $\mathcal{M}'$ of the challenge message $\mathcal{M}$ with respect to the inner hash function. Note that once X_i is known, the following intermediate hash values X_j with $i \leq j \leq 2^s$ are also known. Then, previous generic second-preimage attacks [10] can be applied to find $\mathcal{M}'$ and the complexity is known to be upper-bounded by $O(2^{l-s})$.

6. (online) Query $\mathcal{M}'$ to `MAC` and receive the tag T. Output T as the valid tag for the challenge message $\mathcal{M}$. The complexity of this step is obviously upper-bounded by the block length of $\mathcal{M}'$, that is $O(2^s)$.

Note that there are in total $2^{l-s_1}/l$ nodes in Y, and 2^{s_1} intermediate hash values in X. So a collision between an element in X and an element in Y occurs with a probability around $1/l$. Thus, we need to repeat the attack procedure $\Theta(l)$ times in order to increase the success probability to a constant value.

Now, we can eventually summarize the complexity of the entire universal forgery attack. We recall that $s_1 \leq s-1$.

Step 1:	$O(2^{s_1+l/2})$	**Step 2:**	$O(2^{s_1+l/2})$	**Step 3:**	$O(2^{l-s_1})$
Step 4:	$O(2^{s_1+l/2})$	**Step 5:**	$O(2^{l-s})$	**Step 6:**	$O(2^s)$

- For the case $0 < s < l/4$, the complexity is dominated by Step 3. Set $s_1 = s-1$ and then the total complexity is upper-bounded by $O(l \cdot 2^{l-s})$.
- For the case $l/4 \leq s \leq 3l/4$, set $s_1 = l/4$ to make the complexities at Steps 1 and 3 equal, which optimizes the overall complexity. The complexity is upper-bounded by $O(l \cdot 2^{3l/4})$.
- For the case $s > 3l/4$, set $s_1 = l/4$, and the complexity is dominated by Step 6, which is upper-bounded by $O(l \cdot 2^s)$.

Overall, our attacks have significantly decreased the complexity of universal forgery attack on iterated hash-based MACs from $O(2^{5l/6})$ (attack complexity in [16]) to $O(2^{3l/4})$ by ignoring the polynomial factors.

6 Time-Memory tradeoff for key recovery attacks

In this section, we discuss time-memory tradeoff for key recovery attacks on NMAC or for the equivalent key recovery attacks on HMAC. To start with, it has been known that the complexity of the brute-force key recovery attack can be reduced to 2^l although the key size is $2l$ bits, by following a divide-and-conquer approach. In short, the adversary firstly generates an inner collision, and then brute force recovers the inner key by detecting if the collision can be reached for each key candidate. After the inner key is recovered, the adversary moves to recover the outer key by using the trivial brute-force attack based on the knowledge of the inner key. While this attack does not use any precomputation, surprisingly it is even more efficient than the straightforward application of Hellman's time-memory tradeoff [9], which uses 2^{2l} precomputation, and for key recovery phase $2^{4l/3}$ computations and $2^{4l/3}$ memory. This motivates us to investigate if there are more efficient time-memory tradeoff for the key recovery attacks on HMAC and NMAC with the usage of precomputation.

In the following, we will present our new time-memory tradeoff. Roughly speaking, our tradeoff utilizes both the divide-and-conquer approach and the Hellman's time-memory tradeoff. With a precomputation, we firstly recover the outer key K_{out} and then recover the inner key K_{in}. It is important to note that both of the precomputation for K_{out} and K_{in} are performed before launching any key recovery attacks for K_{out} and K_{in}.

6.1 Recovering K_{out}

Hellman's tradeoff approach is not trivially applicable to recover K_{out} because the input from the inner hash function is unknown due to the inner key. To overcome this problem, we preset the output of the inner hash function to a constant X_e. Thanks to the recent internal state recovery attack on hash-based MAC [12] and the second preimage attack on hash function [10], we can always successfully constructed a message which will produce an output of the inner hash function, which is the fixed X_e.

The attack procedure is described as below.

Precomputation Phase

1. Randomly pick a chaining value X_0 and iteratively compute $X_i = f_M(X_{i-1})$ for $i = 1, \ldots, O(2^{l/2})$ while storing all the X_i's in a lookup table. Denote the final internal state value as X_e.

2. Build Hellman's precomputed lookup tables for the function f_{X_e}, *i.e.*, the compression function with X_e as the message block.

Key Recovery Phase

1. Recover the unknown internal state for a message m with $O(2^{l/2})$ blocks using the technique from [12] with $2^{l/2}$ time complexity and $2^{l/2}$ memory requirement.

2. Append m with an expandable message M_E of range $[l/2, l/2 + 2^{l/2} - 1]$.

3. Find a message block M_L that links the expandable message to one of the precomputed X_i's.

4. Query the MAC oracle with message $\mathcal{M}_q = m \| M_E \| M_L \| M \| \cdots \| M$ to obtain the tag T, where M_E's length is chosen in the way that the overall length of $\mathcal{M}_q$ becomes $O(2^{l/2})$. Note that we shall choose the last block M so that M_q is already a valid padded message, and this message $\mathcal{M}_q$ ensures that the output of the inner layer will be X_e.

5. Use T as the input of Hellman's key recovery phase to recover K_{out}.

In this attack, the first step of the precomputation phase and the second and third steps of the key recovery phase are essentially performed to find a second-preimage of the hash function for the given message $M \| \cdots \| M$ with prefix m, with length $2^{l/3}$ and with the initial value changed to X_0. The entire process costs $2^{2l/3}$ computation and $2^{l/3}$ memory. The second step of the precomputation phase and the fifth step of the key recovery phase are exactly Hellman's tradeoff costing 2^l precomputation, and $2^{2l/3}$ online computations and memory. Overall, K_{out} can be recovered with $2^{2l/3}$ time and $2^{2l/3}$ memory (both dominated by the fifth step) with 2^l precomputation.

6.2 Recovering K_{in}

Our time-memory tradeoff for recovering K_{in} is based on the height of nodes in the functional graph. In short, during the precomputation phase, we collect a set of nodes in a functional graph $\mathcal{G}_{f_M}$ with a certain pattern of heights. Then during the key recovery phase, we first recover the height of K_{in} in $\mathcal{G}_{f_M}$ following the procedure in [16], then derive a set of nodes, which have the same height with K_{in}, from the collected nodes of the prcomputation phase, and checks if K_{in} is inside these nodes or not. Moreover, we need to utilize more than one functional graph in order to amplify the success probability to a constant value.

The attack procedure is described as below.

Precomputation Phase

1. Randomly pick an internal state value X_0 and iteratively compute $X_j = f_{M_i}(X_{j-1})$ until some X_j collides with a previous one. This allows to deduce the height of X_0 in the functional graph $\mathcal{G}_{f_{M_i}}$. Store in table T_i the pair $(X_j, \lambda(X_j))$ with $\lambda(X_j)$ being a multiple of $2^{l/4}$ and $\lambda(X_j) < 2^{l/2}/l$ (omit if the pair is already in T_i). Repeat the process for $2^{l/4}$ random X_0 and sort the table T_i according to the heights, and save the final T_i together with M_i.

2. Repeat the process for random M_i so as to obtain $l \times 2^{l/4}$ structures of (T_i, M_i)'s.

Key Recovery Phase

1. Obtain the height of K_{in} using the technique from [16] using the functional graph $\mathcal{G}_{f_{M_i}}$. Let λ be the smallest multiple of $2^{l/4}$ greater than $\lambda(K_{in})$. Retrieve all X_j's whose height in $\mathcal{G}_{f_{M_i}}$ is equal to λ. Test if $f^{\lambda - \lambda(K_{in})}(X_j)$ is the correct guess of K_{in} for all X_j in the collection of T_i. Repeat for all M_i until K_{in} is recovered.

Following *Conjecture* 1, in the range that interests us (*i.e.*, $[1, 2^{l/2}/l]$), there will be $\Theta(2^{l/4})$ nodes with the same height collected in each table. Since the overall number of nodes at each height of interest is $O(2^{l/2})$, the chance for a collision to happen at each height is $o(2^{-l/4} = 2^{l/4}/2^{l/2})$, and we covered $1/l$ portion of all possible nodes, so the chance to find a match in one table is $o(l^{-1} \cdot 2^{-l/4})$. Since there are $l \cdot 2^{l/4}$ independent tables, our key recovery phase will be successful with a non-negligible probability. The time and memory complexity for this attack is eventually $2^{3l/4}$ with 2^l precomputation.

7 Conclusion

In this paper, we presented selective forgery attacks, improved universal forgery attacks, and time-memory tradeoff for key recovery attacks against the most popular `MAC` constructions built upon iterative hash functions, such as `HMAC` and `NMAC`. Our cryptanalysis methods are based on the extension of various techniques including expandable messages, second-preimage attack, functional graph-based forgery attacks, etc. Our work provides the community with a better understanding of the security margin of iterative hash-based `MAC`s.

Acknowledgments. The authors would like to thank the anonymous referees for their helpful comments, especially for suggesting the conversion from the previous distinguishing-H attack into the selective forgery attack. Jian Guo, Thomas Peyrin and Lei Wang were supported in part by the Singapore National Research Foundation Fellowship 2012 (NRF-NRFF2012-06).

References

1. Bellare, M., Canetti, R., Krawczyk, H.: Keying Hash Functions for Message Authentication. In Koblitz, N., ed.: CRYPTO. Volume 1109 of LNCS., Springer (1996) 1–15
2. Dean, R.D.: Formal Aspects of Mobile Code Security. Ph.D Dissertation, Princeton University (January 1999)
3. Dinur, I., Leurent, G.: Improved Generic Attacks Against Hash-based MACs and HAIFA. In Garay, J., Gennaro, R., eds.: CRYPTO. LNCS, Springer (2014)
4. Dodis, Y., Ristenpart, T., Steinberger, J.P., Tessaro, S.: To Hash or Not to Hash Again? (In)Differentiability Results for H^2 and HMAC. In Safavi-Naini, R., Canetti, R., eds.: CRYPTO. Volume 7417 of LNCS., Springer (2012) 348–366
5. Flajolet, P., Odlyzko, A.M.: Random Mapping Statistics. In Quisquater, J.J., Vandewalle, J., eds.: EUROCRYPT. Volume 434 of LNCS., Springer (1989) 329–354
6. Flajolet, P., Sedgewick, R.: Analytic Combinatorics. Cambridge University Press (2009)
7. Guo, J., Sasaki, Y., Wang, L., Wang, M., Wen, L.: Equivalent Key Recovery Attacks against HMAC and NMAC with Whirlpool Reduced to 7 Rounds. In Cid, C., Rechberger, C., eds.: Fast Software Encryption. LNCS, Springer (2014) to appear.
8. Guo, J., Sasaki, Y., Wang, L., Wu, S.: Cryptanalysis of HMAC/NMAC-Whirlpool. [18] 21–40
9. Hellman, M.E.: A Cryptanalytic Time-Memory Trade-Off. IEEE Transactions on Information Theory **26**(4) (1980) 401–406
10. Kelsey, J., Schneier, B.: Second Preimages on n-Bit Hash Functions for Much Less Than 2^n Work. In Cramer, R., ed.: EUROCRYPT. Volume 3494 of LNCS. Springer (2005) 474–490
11. Krawczyk, H., Bellare, M., Canetti, R.: HMAC: Keyed-Hashing for Message Authentication. Internet Engineering Task Force (IETF). (1997) `http://www.rfc-editor.org/rfc/rfc2104.txt`.
12. Leurent, G., Peyrin, T., Wang, L.: New Generic Attacks against Hash-Based MACs. [18] 1–20
13. Mutafchiev, L.R.: The limit distribution of the number of nodes in low strata of a random mapping. Statistics & probability letters **7**(3) (1988) 247–251
14. Naito, Y., Sasaki, Y., Wang, L., Yasuda, K.: Generic State-Recovery and Forgery Attacks on ChopMD-MAC and on NMAC/HMAC. In Sakiyama, K., Terada, M., eds.: IWSEC. Volume 8231 of LNCS., Springer (2013) 83–98
15. Peyrin, T., Sasaki, Y., Wang, L.: Generic Related-Key Attacks for HMAC. In Wang, X., Sako, K., eds.: ASIACRYPT. Volume 7658 of LNCS., Springer (2012) 580–597
16. Peyrin, T., Wang, L.: Generic Universal Forgery Attack on Iterative Hash-Based MACs. In Nguyen, P.Q., Oswald, E., eds.: EUROCRYPT. Volume 8441 of LNCS., Springer (2014) 147–164
17. Preneel, B., van Oorschot, P.C.: On the Security of Two MAC Algorithms. In Maurer, U.M., ed.: EUROCRYPT. Volume 1070 of LNCS., Springer (1996) 19–32
18. Sako, K., Sarkar, P., eds.: Advances in Cryptology - ASIACRYPT 2013 - 19th International Conference on the Theory and Application of Cryptology and Information Security, Bengaluru, India, December 1-5, 2013, Proceedings, Part II. In Sako, K., Sarkar, P., eds.: ASIACRYPT (2). Volume 8270 of LNCS., Springer (2013)
19. SBI Net Systems: MonoCrypt home page. `http://capg.sbins.co.jp/products/monocrypt/index.html`.
20. U.S. Department of Commerce, National Institute of Standards and Technology: Secure Hash Standard (SHS) (Federal Information Processing Standards Publication 180-3). (2008) `http://csrc.nist.gov/publications/fips/fips180-3/fips180-3\_final.pdf`.

21. U.S. Department of Commerce, National Institute of Standards and Technology: Recommendation for Applications Using Approved Hash Algorithms (Federal Information Processing Standards Publication 800-107). (2012) `http://csrc.nist.gov/publications/nistpubs/800-107-rev1/sp800-107-rev1.pdf`.
22. Yasuda, K.: ”Sandwich” Is Indeed Secure: How to Authenticate a Message with Just One Hashing. In Pieprzyk, J., Ghodosi, H., Dawson, E., eds.: ACISP. Volume 4586 of LNCS., Springer (2007) 355–369