Generic Universal Forgery Attack on Iterative Hash-based MACs

Thomas Peyrin and Lei Wang

Division of Mathematical Sciences, School of Physical and Mathematical Sciences,
Nanyang Technological University, Singapore
thomas.peyrin@gmail.com wang.lei@ntu.edu.sg

Abstract. In this article, we study the security of iterative hash-based MACs, such as HMAC or NMAC, with regards to universal forgery attacks. Leveraging recent advances in the analysis of functional graphs built from the iteration of HMAC or NMAC, we exhibit the very first generic universal forgery attack against hash-based MACs. In particular, our work implies that the universal forgery resistance of an n-bit output HMAC construction is not 2^n queries as long believed by the community. The techniques we introduce extend the previous functional graphs-based attacks that only took in account the cycle structure or the collision probability: we show that one can extract much more meaningful secret information by also analyzing the distance of a node from the cycle of its component in the functional graph.

Key words: HMAC, NMAC, hash function, universal forgery

1 Introduction

A message authentication code (MAC) is a crucial symmetric-key cryptographic primitive, which provides both authenticity and integrity for messages. It takes a k-bit secret key K and an arbitrary long message M as inputs, and produces an n-bit tag. In the classical scenario, the sender sends both a message M and a tag $T = \mathtt{MAC}(K, M)$ to the receiver, where the secret key K is shared between the sender and the receiver prior to the communication. Then, the receiver computes another tag value $T' = \mathtt{MAC}(K, M)$ using her own key K, and matches T' to the received T. If a match occurs, the receiver is ensured that M was indeed sent by the sender and has not been tampered with by a third party.

There are several ways to build a MAC from other symmetric-key cryptographic primitives, but a very popular approach is to use a hash function. In particular, a well-known example is HMAC [2], designed by Bellare, Canetti and Krawczyk in 1996. HMAC has been internationally standardized by ANSI, IETF, ISO and NIST, and is widely implemented in various worldwide security protocols such as SSL, TLS, IPSec, etc.

Being cryptographic objects, MACs should satisfy various security requirements and the classical notions are key recovery resistance and unforgeability:

- *Key recovery resistance*: it should be practically infeasible for an adversary to recover the value of the secret key.
- *Unforgeability*: it should be practically infeasible for an adversary to generate a message and tag pair (M, T) such that T is a valid tag for M and such that M has not been queried to `MAC` previously by the adversary.

In the case of an ideal `MAC`, the attacker should not be able to recover the key in less than 2^k computations, nor to forge a valid `MAC` in less than 2^n computations. Depending on the control of the attacker over the message, one discriminates between two types of forgery attacks: existential forgery and universal forgery attack. In the former case, the attacker can fully choose the message M for which he will forge a valid tag T, while in the later case he will be challenged to a certain message M and must find the `MAC` tag value T for this particular message. In other words, universal forgery asks the attacker to be able to forge a valid `MAC` on any message, and as such is a much more powerful attack than existential forgery and would lead to much more damaging effects in practice. Yet, because this security notion is easier to break, most published attacks on `MAC`s concern existential forgery.

Moreover, cryptographers have also proposed a few extra security notions with respect to distinguishing games such as distinguishing-R and distinguishing-H [12]. The goal of a distinguishing-R attack is to distinguish a `MAC` scheme from a monolithic random oracle, while the goal of a distinguishing-H attack is to distinguish hash function-based `MAC`s (resp. block cipher and operating mode-based `MAC`s) instantiated with either a known dedicated compression function (resp. a dedicated block cipher) or a random function (resp. a random block cipher). While these distinguishers provide better understanding of the security margin, the impact to the practical security of a `MAC` scheme would be rather limited.

Given the importance of `HMAC` in practice, it is only natural that many researchers have analyzed the security of this algorithm and of hash-based `MAC`s in general. On one hand, cryptographers are devoted to find reduction-based security proofs to provide lower security bound. Usually a `MAC` based on a hash function with a l-bit internal state is proven secure up to the bound $O(2^{l/2})$. Examples include security proofs for `HMAC`, `NMAC` and `Sandwich-MAC` [2, 1, 25]. Namely, it is guaranteed that no generic attack succeeds with a complexity below the security bound $O(2^{l/2})$ (when $l \leq 2n$) in the single-key model.

On the other hand, cryptographers are also continuously searching for generic attacks to get upper security bound for hash-based `MAC`s, since the gap between the $2^{l/2}$ lower bound and the best known generic attacks is still very large for several security properties. The cases of existential forgery and distinguishing-R attacks are tight: in [17], Preneel and van Oorschot proposed generic distinguishing-R and existential forgery attacks with a complexity of $O(2^{l/2})$ computations. Their methods are based on the generation of internal collisions which are detectable on the `MAC` output due to the length extension property of the inner iterated hash function (one can generate an existential forgery by simply look-

ing for an internal collision in the hash chain and then, given any pair of messages using this internal collision as prefix, it is easy to forge the tag for one message by querying the other message to the tag oracle).

In [16] Peyrin *et al.* utilized the cycle property of HMAC in the related-key model to distinguish it from a random mapping and eventually described generic distinguishing-R attack with a complexity of only $O(2^{n/2})$ computations (note that these related-key attacks do not contradict the $O(2^{l/2})$ security proof which was provided in the single-key model only). A similar weakness was independently pointed out by Dodis *et al.* in the context of indifferentiabiity of HMAC [5]. One year after, leveraging the ideas of cycle detection in functional graphs from [16], Leurent *et al.* [14] showed that, contrary to the community belief, there exists a generic distinguishing-H attack requiring only $O(2^{l/2})$ computations on iterative hash-based MACs in the single-key model. All security bounds on iterative hash-based MACs are therefore tight, except the case of universal forgery for which the best generic attack still requires 2^n computations and it remains unknown exactly where the security lies between 2^n and $\min\{2^{l/2}, 2^n\}$ computations.

Besides generic attacks, cryptanalysts also evaluated MACs based on (standardized) dedicated hash functions, mainly by exploiting some weakness of the compression function [3, 12, 8, 23, 19, 20, 13, 24, 26, 22, 9]. The details of such attacks will be omitted in the rest of this article, since we deal with generic attacks irrespective to the specifications of the internal compression function.

Our contribution. In this article, we describe the first generic universal forgery attack on iterative hash-based MACs, requiring less then 2^n computations. More precisely, our attack complexity is $O(\max(2^{l-s}, 2^{5l/6}, 2^s))$, where 2^s represents the block length of the challenge message. In other words, for reasonable message sizes, the complexity directly decreases along with an increase of s, up to a message size of $2^{l/6}$ where the complexity hits a plateau at $2^{5l/6}$ computations. Previously known attacks and proven bounds are summarized in Table 1 and we emphasize that this is the first generic universal forgery attack on HMAC in the single key model (except the trivial 2^n brute force attack). For example, a corollary to our work is that HMAC instantiated with the standardized hash function RIPEMD-160 [4] (or MD5 [21] and RIPEMD-128 [4]), which allows arbitrarily long input messages (this conditions is needed since even though the challenge message can have a small length, we will need to be able to query $2^{l/2}$-block long messages during the attack), only provides a $2^{133.3}$ (resp. $2^{106.7}$) computations security with regards to universal forgery attacks, while it was long believed that the full 2^{160} (resp. 2^{128}) was holding for this strong security property.

Moreover, our techniques are novel as they show that one can extract much more meaningful secret information than by just analyzing the cycle structure or the collision probability of the functional graphs of the MAC algorithm, as was done previously [16, 14]. Indeed, the distance of a node from the cycle of its components in the functional graph is a very valuable information to know

Table 1. We summarize the security state of HMAC (with $n \leq l \leq 2n$) including previous results and our universal forgery attacks. Notation max() is to choose the largest value.

security notion	single key setting		related-key setting
	provable security	generic attack	generic attack
Distinguishing-R	$O(2^{l/2})$ [2, 1]	$O(2^{l/2})$ [17]	$O(2^{n/2})$ [16]
Distinguishing-H	$O(2^{l/2})$ [2, 1]	$O(2^{l/2})$ [14]	$O(2^{n/2} + 2^{l-n})$ † [16]
Existential forgery	$O(2^{l/2})$ [2, 1]	$O(2^{l/2})$ [17]	$O(2^{n/2} + 2^{l-n})$ † [16]
Universal forgery	$O(2^{l/2})$ [2, 1]	**previous:** $O(2^n)$ **new:** $O(\max(2^{l-s}, 2^{5l/6}, 2^s))$ ‡	

†: the attacks have complexity advantage with $n < l < 2n$;
‡: 2^s is the blocks length of the challenge message. The attack has complexity advantage with $n \leq l < 6n/5$.

for an attacker, and we expect even more complex types of information to be exploitable by attackers against iterative hash-based MACs.

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 fixed input length compression function f, which maps inputs of $l+b$ bits to outputs of l bits (note that $l \geq n$). In details, H first pads an input message M to be a multiple of b bits, then splits it into blocks of b bits $m_0||m_1||\cdots||m_{s-1}$, and calls the compression function f iteratively to process these blocks. Finally, H might use a finalization function g that maps l bits to n bits in order to produce the hash digest.

$$x_0 = IV \qquad x_{i+1} = f(x_i, m_i) \qquad \texttt{hashdigest} = g(x_s)$$

Each of the chaining variables x_i are l bits long, and IV (initial value) is a public constant.

NMAC algorithm [2] 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 M as:

$$\texttt{NMAC}(K_{out}, K_{in}, M) = H_{K_{out}}(H_{K_{in}}(M)).$$

The keyed hash functions $H_{K_{in}}$ and $H_{K_{out}}$ are referred to as the inner and the outer hash functions respectively.

HMAC algorithm [2] 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:

$$K_{in} = f(IV, K \oplus \texttt{ipad}) \qquad K_{out} = f(IV, K \oplus \texttt{opad})$$

where ipad and opad are two distinct public constants. HMAC is then simply defined to process an input message M as:

$$\texttt{HMAC}(K, M) = H(K \oplus \texttt{opad} \| H(K \oplus \texttt{ipad} \| M)))$$

where $\|$ denotes the concatenation operation. It is interesting to note that HMAC can use 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 an entire message block. Otherwise, if the key K is longer than b bits, then it is hashed and then padded with 0 bits: $K \leftarrow H(K) \| 0^{b-n}$.

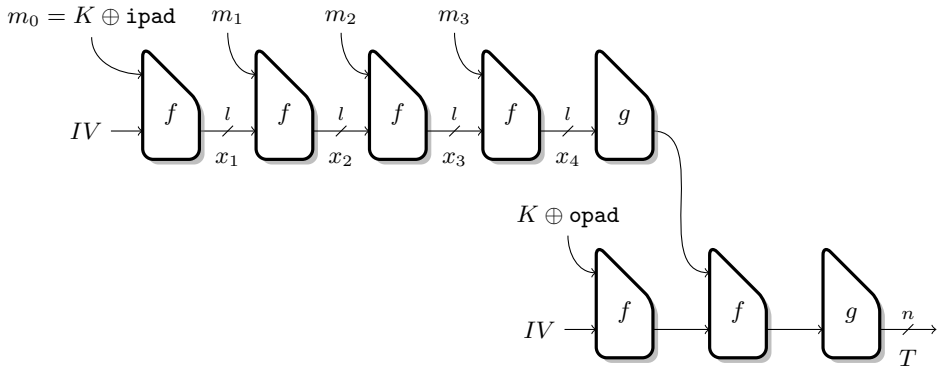

Fig. 1. HMAC with an iterated hash function with compression function f, and output function g.

For simplicity, in the rest of this article we will describe the attacks based on the utilization of the HMAC algorithm. However, we emphasize that our methods apply similarly to hash-based MACs such as NMAC [2], Sandwich-MAC [25], etc.

3 Previous functional-graph-based attacks for HMAC

Our universal forgery attack is based on recent advances in hash-based MACs cryptanalysis [16, 14] and in this section we quickly recall these methods and explain how we extend them. First of all, we need to introduce the notion of functional graph and the various properties that can be observed from it.

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 iff $f(a) = b$). The functional graph of a function is composed of one or several components, each having its own internal cycle.

For a random function, the functional graph will possess several statistical properties that have been extensively studied. For example, it is to be noted that with high probability the functional graph of a random function will have a logarithmic number of components and among them there is one giant component that covers most of the nodes. In addition, this giant component will contain a giant tree in which are present about a third of the nodes of $\mathcal{G}_f$. Theorems 1 and 2 state these remarks in a more formal way.

Theorem 1 ([6, 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 ([6, 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 [7].

Theorem 3 ([7, 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* $0.7582 * N$.

Knowing all these statistical properties for the functional graph of a random function, Peyrin *et al.* [16] studied the successive iterations of `HMAC` with a fixed small message block for two related-keys K and $K = K \oplus \texttt{ipad} \oplus \texttt{opad}$. Thanks to a small weakness of `HMAC` in the related-key setting, they observed that the two corresponding functional graphs are exactly the same (while ideally they should look like the functional graphs of two independent random functions)

and this can be detected on the output of HMAC by measuring the cycle lengths. They used this property to derive generic distinguishing-R, distinguishing-H and existential forgery attacks in the related-key setting.

Later, Leurent *et al.* [14] extended the scope of cycle detection by providing a single-key utilization of this technique. Namely, they show how to craft two special long messages (mainly composed of identical message blocks), both following two separate cycle loops in the functional graph of the internal compression function. This trick allows the two messages to collide after the last processed message block, but also to have the same length (and thus the processing of the final padding block would not reintroduce differences). Such a collision can therefore be detected on the output of HMAC, and they use this special information leakage (information is leaked on the unknown internal compression function used) to derive a generic distinguishing-H attack in the single-key setting. They also provide another attack that can trade extra complexity cost for smaller message size, and in which the property scrutinized is the probability distribution of the collisions in the functional graph.

From a high-level perspective, these two previous works mainly considered as distinguishing properties the cycle nodes or the collisions distribution in a functional graph. In this article, we consider a functional graph property which seems not trivial to exploit: the height λ of a tail node, i.e. the distance of a node from the cycle of its component. While not trivial and likely to be costly, the potential outcome of analyzing such a property is that if one can extract this information leakage from the HMAC output, he would get direct information on a particular node of the computation. The attack can therefore be much sharper (the size of the cycle is not a powerful property as it represents a footprint equivalent for all the nodes of the component, while the height of a node is much more discriminating), and that is the reason why it eventually allows us to derive a generic universal forgery attack in the classical single-key setting.

4 General description of the universal forgery attack

Let $M_t = m_1 \| m_2 \| \dots \| m_s$ be the target message to forge given by the challenger to the adversary (we start the counting from m_1 since the first message block m_0 to be processed by the inner hash function call is $m_0 = K \oplus \texttt{ipad}$). In order to forge the tag value corresponding to this message, we will construct a different message M'_t which will collide with M_t in the inner hash function of HMAC, namely $H_{K_{in}}(M'_t) = H_{K_{in}}(M_t)$, and this directly leads to colliding tags on the output of the HMAC: $T = H_{K_{out}}(H_{K_{in}}(M'_t)) = H_{K_{out}}(H_{K_{in}}(M_t))$. Then, by simply querying the HMAC value T of M'_t, we eventually forge a valid tag corresponding to M_t by outputting T.

Constructing such a message M'_t is in fact equivalent to finding a second preimage of M_t on the keyed hash function $H_{K_{in}}$. While second preimage attacks have been published on public iterative hash functions [11], unfortunately they cannot be applied to a keyed hash function as they depend on the knowledge of the intermediate hash values when processing M_t. However, in our situation the

intermediate hash values for $H_{K_{in}}(M_t)$ are hidden since only the tag is given as output and since K_{in} is unknown to the adversary, and so he will not be able to guess them. We will overcome this issue by proposing a novel approach to recover some intermediate hash value x_i from the computation of $H_{K_{in}}(M_t)$. We stress that this is different from and much harder than previous so-called internal state recovery in [14], which recovers some internal state of a message completely chosen by the adversary himself. Note that once x_i is recovered, we get to know all the next intermediate hash values by simply computing $x_{i+1} = f(x_i, m_i), \ldots, x_{s+1} = f(x_s, m_s)$ since H is an iterative hash function. Once these intermediate hash values are known, we can apply the previous second preimage attacks [11] in order to find M'_t.

In order to recover one value from the set of the intermediate chaining values $X = \{x_1, x_2, \ldots, x_{s+1}\}$ of $H_{K_{in}}(M)$, we choose offline $2^l/s$ values $Y = \{y_1, y_2, \ldots, y_{2^l/s}\}$, and one can see that with a good probability one element y_j of Y will collide with an element in X. We need to filter out this y_j value and this seems not easy since there is no previously published suitable property on the intermediate hash values of `HMAC` that the adversary can detect on the output.

One may consider using internal collisions, which are detectable by searching for colliding tags due to the length extension property: finding a message pair (m, m') for x_i such that $f(x_i, m) = f(x_i, m')$ by querying `HMAC` online and then using this pair to determine if $y_j = x_i$ holds by checking offline if $f(y_j, m) = f(y_j, m')$ holds. However, note that with this naive method only a single x_i can be tested at a time (since other $x_{i'}$ with $i' \neq i$ are very likely not to collide with the message pair (m, m')) and we will therefore have to repeat this procedure for each value of X independently. Thus, this attack fails as we would end up testing 2^l pairs and reaching a too high complexity.

Overall, it is essential to find a new property on the intermediate hash values of `HMAC` such that it can be detected by the adversary and such that it can be exploited to match a value of Y to all the values in X **simultaneously**. In our attack, we will use a novel property, yet unexploited: the height $\lambda(x_i)$ of each x_i of X in the functional graph of f_V, where f_V stands for the compression function with the message block fixed to a value V; $f_V(\cdot) = f(\cdot, V)$. In the rest of this article, without loss of generality, we will let V be the message block only composed of zero bits and we denote $f_{[0]}(\cdot) = f(\cdot, [0])$ the corresponding compression function.

4.1 The height property of a node in a functional graph

In the functional graph of a random mapping on a finite set of size N, it is easy to see that each node x has a unique path connecting it with a cycle node, and we denoted the length of this path the **height** $\lambda(x)$ of x (or **tail length**). Obviously, for cycle nodes, we have $\lambda = 0$. 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. Researchers have carried out extensive studies on the distribution of S_λ as $N \to \infty$. In particular, Harris proved that the mean value of S_0 is $\sqrt{\pi N/2}$ [10],

which is consistent with Theorem 1 as the number of the cycle nodes. After that, Mutafchiev [15] proved the following theorem as an extension of Harris's result.

Theorem 4 ([15, Lemma 2]). *If $N \to \infty$ and $\lambda = o(\sqrt{N})$, the mean value of the λ-th stratum S_λ is $\sqrt{\pi N/2}$.*

Note that Mutafchiev's result is no longer true for $\lambda = O(\sqrt{N})$ and, for interested readers, we refer to [18] for the limit distribution of S_λ with $\lambda = O(\sqrt{N})$.

Interestingly, if the largest component is removed from the functional graph, then the remaining components also form a functional graph of a random mapping on a finite set of size $(1 - 0.7582) * N = 0.2418 * N$ (since Theorem 3 tells us that the largest component has an expected number of nodes of $0.7582 * N$). Thus we get the following corollary.

Corollary 1. *If $N \to \infty$ and $\lambda = o(\sqrt{N})$, the mean value of the λ-th stratum S_λ in the largest component is $0.64\sqrt{N} = \sqrt{\pi N/2} - \sqrt{\pi N * 0.2418/2}$.*

Now we move back to discuss about the height distribution in the functional graph $\mathcal{G}_{f_{[0]}}$ of $f_{[0]}$. From Corollary 1, we can deduce that if $l \to \infty$ and $\lambda = o(2^{l/2})$, the mean value of S_λ in the largest component of $f_{[0]}$ is $0.64 * 2^{l/2}$. In order to illustrate the notion of $\lambda = o(2^{l/2})$ more clearly, we rewrite the corollary into the following equivalent one.

Corollary 2. *Let $\delta(l)$ be any function such that $\delta(l) \to \infty$ as $l \to \infty$. There exists a positive value l_0 such that for any $l > l_0$, the mean value of λ-th stratum S_λ with $0 \leq \lambda \leq 2^{l/2}/\delta(l)$ in the largest component is $0.64 * 2^{l/2}$.*

Next, we will utilize Corollary 2 to prove the lower bound on the number of distinct height values of the intermediate chaining values in X, which we will use in order to evaluate the attack complexity. Denote the set of all the nodes with a height $\lambda \in [0, 2^{l/2}/\delta(l)]$ as N', which covers in total $0.64 * 2^l/\delta(l)$ nodes. Thus, a random node belongs to N' with a probability $0.64/\delta(l)$. Moreover, from Corollary 2, for a random node in N', its height is uniformly distributed in $[0, 2^{l/2}/\delta(l)]$. From these properties of N', we get that $0.64 * s/\delta(l)$ elements in X belong to N'. Moreover, there is no collision on the height among these elements with an overwhelming probability if $s \ll 2^{l/4}$ holds. Note that in our forgery attack, we will set s to be at most $2^{l/6}$ (see Section 5.1 for the details). Overall, the lower bound on the number of distinct height values in X is $0.64 * s/\delta(l)$. It is important and interesting to note that from Corollary 2, if l becomes very large, $\delta(l)$ will become negligible compared to exponential-order computations $2^{\Omega(l)}$, e.g., $\delta(l) = \log(l)$.

On the other hand, we performed experiments to evaluate the expected number of the distinct height values in X. More precisely, we used `SHA-256` compression function for small values of l. We prepend 0^{256-l} to a l-bit value x, then compute y=`SHA-256` $(0^{256-l} \| x)$, and finally output the l LSBs of y. With $l \leq 30$, we generated random pairs and checked if their heights collide or not

in the functional graph of l-bit truncated SHA-256 compression function. The experimental results show that a pair of random values has a colliding height with a probability of around $2^{-l/2}$. Moreover, it is matched with a rough probability estimation as follows. Let x and x' be two randomly chosen l-bit values. Suppose x and x' have the same height, then it implies that after i iterations of $f_{[0]}$ (denoted as $f^i_{[0]}$), either one of the following two cases occurs. One is $f^i_{[0]}(x) = f^i_{[0]}(x')$, which has a probability of roughly 2^{-l} for each i conditioned on $f^{i-1}_{[0]}(x) \neq f^{i-1}_{[0]}(x')$. The other one is that $f^i_{[0]}(x) \neq f^i_{[0]}(x')$ and both $f^i_{[0]}(x)$ and $f^i_{[0]}(x')$ enter the component cycle simultaneously, which has a probability of roughly $(\sqrt{\pi/2} * 2^{-l/2})^2 = \pi/2 * 2^{-l}$ for each i, since the number of cycle nodes is $\sqrt{\pi/2} * 2^{l/2}$. Note that Theorem 2 proved the expected tail length is $\sqrt{\pi/8} * 2^{l/2}$. Thus, if neither of the two cases occurs up to $\sqrt{\pi/8} * 2^{l/2}$ iterations, we get that $f^i(x)$ and $f^{i'}(x')$ enter the component cycle with different i and i', namely x and x' have different heights. So the total probability of randomly chosen x and x' having the same height is at most $2^{-l} * \sqrt{\pi/8} * 2^{l/2} + \pi/2 * 2^{-l} * \sqrt{\pi/8} * 2^{l/2} \approx 2^{-l/2}$. Overall, we make a natural, conservative and confident conjecture as follows (note that s is at most $2^{l/6}$ in our attacks. See Section 5.1 for the details).

Conjecture 1. With $s \leq 2^{l/6}$, there is only a negligible probability that a collision exists among the heights of s random values in a functional graph of a l-bit random mapping.

In the rest of the paper, we will describe our attacks based on the Conjecture 1, namely the heights of the intermediate hash values in X are distinct. However, if only taking in account the proven lower bound $0.64 * s/\delta(l)$ of the number of the distinct heights in X, the number of offline nodes should be increased by $\delta(l)/0.64$ times, and the attack complexity is increased by a factor of $O(\delta(l))$. Note that $O(\delta(l))$ is negligible compared to $2^{O(l)}$, and thus it has very limited influence to the complexity for large l.

4.2 Deducing online the height of a few intermediate hash values

We now explain how to deduce the height $\lambda(x_i)$ of a node x_i in the functional graph $\mathcal{G}_{f_{[0]}}$ of $f_{[0]}$. We start by finding the cycle length of the largest component of $\mathcal{G}_{f_{[0]}}$, and we denote it by L. This can be done offline with a complexity of $O(2^{l/2})$ computations, as explained in [16]. Then, we ask for the MAC computation of two messages M_1 and M_2:

$$M_1 = m_1 \| m_2 \| \dots \| m_{i-1} \| [0]^{2^{l/2}+L} \| [1] \| [0]^{2^{l/2}}$$
$$M_2 = m_1 \| m_2 \| \dots \| m_{i-1} \| [0]^{2^{l/2}} \| [1] \| [0]^{2^{l/2}+L}$$

where $[0]^j$ represents j consecutive zero-bit message blocks, and we check if the two tags collide. It is important to note that if the intermediate hash value x_i is located in the largest component of $\mathcal{G}_{f_{[0]}}$ and has a height $\lambda(x_i)$ no larger than

$2^{l/2}$, then the intermediate hash value after processing $m_1\|m_2\|\ldots\|m_{i-1}\|[0]^{2^{l/2}}$ is in the cycle of the largest component. Also, the intermediate hash values after processing $m_1\|m_2\|\ldots\|m_{i-1}\|[0]^{2^{l/2}}\|[1]$ and $m_1\|m_2\|\ldots\|m_{i-1}\|[0]^{2^{l/2}+L}\|[1]$ will be equal (and we denote it by x) since in the latter we just make an extra cycle walk before processing the message block [1]. Under a similar reasoning, if x is also in the largest component[1] and has a height $\lambda(x)$ no larger than $2^{l/2}$, we get that the intermediate hash values after processing $m_1\|m_2\|\ldots\|m_{i-1}\|[0]^{2^{l/2}+L}\|[1]\|[0]^{2^{l/2}}$ and $m_1\|m_2\|\ldots\|m_{i-1}\|[0]^{2^{l/2}}\|[1]\|[0]^{2^{l/2}+L}$ are equal. Moreover, since M_1 and M_2 have the same block length, we get a collision on the inner hash function, which directly extends to a collision on the output tag. From the functional graph properties of a random function given in Sections 3 and 4.1, a randomly chosen node will be located in the largest component of $\mathcal{G}_{f_{[0]}}$ with a probability of about 0.7582 and will have a height no larger than $2^{l/2}$ with a probability roughly 0.5. Thus, M_1 and M_2 will collide with a probability $(0.7582 * 0.5)^2 = 0.14$.

In order to recover the height $\lambda(x_i)$ of one node x_i in the functional graph $\mathcal{G}_{f_{[0]}}$ of $f_{[0]}$, we will test $\log(l)$ message pairs obtained from (M_1, M_2) by changing the block [1] to other values. If (at least) one of these pairs collides, we can deduce that with overwhelming probability[2] x_i is in the largest component, and has a height $\lambda(x_i)$ of at most $2^{l/2}$. Otherwise, we give up on recovering the height $\lambda(x_i)$ of x_i, and move to find the height $\lambda(x_{i+1})$ of the next intermediate hash value x_{i+1}.

In the former situation, we can start to search for the exact node height $\lambda(x_i)$ of x_i in $\mathcal{G}_{f_{[0]}}$, and we will accomplish this task thanks to a **binary search**. Namely, we first check whether the intermediate hash value after processing $m_1\|m_2\|\ldots\|m_{i-1}\|[0]^{2^{l/2-1}}$ is in the cycle or not (note that we now have $2^{l/2-1}$ [0] blocks in the middle, instead of $2^{l/2}$ originally), and this can be done by asking for the `MAC` computation of two messages M_1^* and M_2^*

$$M_1^* = m_1\|m_2\|\ldots\|m_{i-1}\|[0]^{2^{l/2-1}}\|[1]\|[0]^{2^{l/2}+L}$$
$$M_2^* = m_1\|m_2\|\ldots\|m_{i-1}\|[0]^{2^{l/2-1}+L}\|[1]\|[0]^{2^{l/2}}$$

and by checking if their respective tags collide. After testing $\log(l)$ such pairs obtained from (M_1^*, M_2^*) by modifying the block [1] to other values, if (at least) one pair collides, we can deduce that with overwhelming probability the intermediate hash value after processing $m_1\|m_2\|\ldots\|m_{i-1}\|[0]^{2^{l/2-1}}$ is in the cycle, and the height $\lambda(x_i)$ of x_i is no larger than $2^{l/2-1}$. Otherwise, we deduce that $\lambda(x_i)$ lies between $2^{l/2-1}$ and $2^{l/2}$. Thus, the amount of possible height values

[1] Since we processed a message block [1], different from [0], the last computation will not follow the functional graph $\mathcal{G}_{f_{[0]}}$ and we will be mapped to a random point in $\mathcal{G}_{f_{[0]}}$.

[2] Since the probability that x_i is in the largest component and has a height $\lambda(x_i) \leq 2^{l/2}$ is constant, choosing $\log(l)$ messages will ensure that the success probability of this step is very close to one, see [14].

for x_i are reduced by one half. We continue iterating this binary search procedure $\log_2(2^{l/2}) = l/2$ times, and we will eventually obtain the exact height value $\lambda(x_i)$ of x_i. By applying such a height recovery procedure, we get to know the height value for $0.38 * s$ values in X on average (one intermediate hash value x_i has probability 0.7582 to be located in the biggest component, and probability about $1/2$ to have a height not greater than $2^{l/2}$).

4.3 Deducing offline the height of many chosen values

Before we start to retrieve a value x_i of X, we need to handle the set Y offline. When we choose values to build the set $Y = \{y_1, y_2, \ldots, y_{2^l/s}\}$, we also have to compute their respective height in the functional graph of $f_{[0]}$. One may consider to use a trivial and random sampling, i.e. choosing random nodes first and then computing their height. Note that such a procedure is very expensive, since computing height for a random value requires around $2^{l/2}$ computations on average, which renders the total complexity of building Y beyond 2^l. We propose instead to use an offline sampling procedure as follows.

We first initialize Y as an empty set and we start by choosing a new and random value y_1, namely $y_1 \notin Y$. Then, we apply $f_{[0]}$ to update it successively; $y_{i+1} = f(y_i, [0])$, and memorize all the y_i's in the computation chain. The iteration terminates if y_{i+1} collides with a previous stored value $y \in Y$ whose height is already known (we denote it as λ), or if it collides with a previous node y_j $(1 \leq j \leq i)$ in the current chain, namely a new cycle is generated. In the former case, we naturally compute the height of nodes y_p $(1 \leq p \leq i)$ in the chain as $\lambda + i + 1 - p$, and store all of them in Y. For the latter case, we set the height of all the nodes from y_j to y_i as 0 (since they belong to the cycle of their own component), and we then compute the height of tail nodes y_p $(0 \leq p < j)$ as $j - p$ and store all of them in Y.

Using this procedure, we can select $2^l/s$ values and obtain their height with a complexity of only $2^l/s$ computations. Moreover, from the functional graph properties of a random function given in Sections 3 and 4.1, we know that on average 38% values in Y are located in the largest components and have a height no larger than $2^{l/2}$.

Note that Y is not a set of random values. We do not know the distribution of height values of the elements in Y, which essentially makes Conjecture 1 be necessary for our attack. The detailed discussion follows in next section.

4.4 Exploiting the height information leakage

At this point, the attacker built the sets X and Y and knows the height of almost all their elements (for X, only the heights of $0.38 * s$ elements are known). The next step is to recover one value in X (which are still unknown to the attacker) by matching between the elements in set X and the elements in set Y. However, for each x_i in X, we do not have to try to match every value in Y. Indeed, we just need to pay attention to a smaller subset of Y in which the elements have

the same height value as x_i. Moreover, since the elements in the set X have distinct heights (see details in Section 4.1), these subsets of Y are all disjoint. Thus in total we need to match at most $2^l/s$ pairs, namely the size of Y. This point is precisely where the adversary will get a complexity advantage during his attack.

4.5 Attack summary

Finally, let us wrap everything up and describe the universal forgery attack from the very beginning. The adversary is given a target message $M_t = m_1\|m_2\|\ldots\|m_s$ by the challenger, for which he has to forge a valid tag. He splits M_t into two parts $M_{t_1}\|M_{t_2}$:

$M_{t_1} = m_1\|m_2\|\ldots\|m_{s_1}$ will be used for the intermediate hash value recovery,
$M_{t_2} = m_{s_1+1}\|m_{s_1+2}\|\ldots\|m_s$ will be used in the second preimage attack.

During the online phase, the adversary applies the height recovery procedure from Section 4.2 for each x_i ($1 \leq i \leq s_1+1$), and stores them in X. Moreover, he produces a filter (m, m') for each x_i such that $f(x_i, m) = f(x_i, m')$ holds. During the offline phase, the adversary chooses $2^l/s_1$ values following the sampling procedure from Section 4.3 and stores them in Y.

Then, he recovers the value of one of the x_i's by matching the sets X and Y: for each x_i, he checks if $f(y, m) = f(y, m')$ holds or not for all y's that have the same height as x_i in Y. If a collision is found, then y is equal to x_i with a good probability. Once one x_i ($1 \leq i \leq s_1+1$) is recovered, the adversary gets to know the value of x_{s_1+1} by computing the iteration $x_{i+1} = f(x_i, m_{i+1}), \ldots, x_{s_1+1} = f(x_{s_1}, m_{s_1})$, which induces that the latter half of the inner hash function when processing M_t is equivalent to a public hash function by regarding x_{s_1+1} as the public IV. Thus, the adversary is able to apply previous second preimage attacks on public hash functions [11] to find a second preimage M'_{t_2} for M_{t_2}. In the end, the adversary queries $M_{t_1}\|M'_{t_2}$ to the `MAC` oracle and receives a tag value T. This tag T is also a valid tag for the challenge M_t and the universal forgery attack succeeds.

5 Full procedure of the universal forgery attack

In this section, we provide the entire procedure of this complex attack and we first recall the notations used. Let $M_t = m_1\|m_2\|\ldots\|m_s$ be the challenge message (we start the counting from m_1, since $m_0 = K \oplus \texttt{ipad}$ during the first compression function call of the inner hash call of `HMAC`) and we denote by $x_1, x_2, \ldots, x_{s+1}$ the successive intermediate hash values of $H_{K_{in}}(M_t)$ when processing M_t. During the attack, M_t is divided into $M_{t_1}\|M_{t_2}$, where M_{t_1} is $m_1\|m_2\|\ldots\|m_{s_1}$ and M_{t_2} is $m_{s_1+1}\|m_{s_1+2}\|\ldots\|m_s$. As an example, we will use the functional graph $\mathcal{G}_{f_{[0]}}$ of the hash compression function f when iterated with a fixed message block $[0]$ and we denote by L the cycle length of the largest component of $\mathcal{G}_{f_{[0]}}$.

Phase 1 (online). Recover the height of x_1, x_2, ..., and x_{s_1+1} in $\mathcal{G}_{f_{[0]}}$ and store them in a set X. The procedure is detailed as below.

1. Initialize an index counter c as 1, and the set X as empty.
2. Query to the MAC oracle and receive the corresponding tag pairs of $\log(l)$ distinct message pairs $m_1\| \ldots \| m_{c-1}\| [0]^{2^{l/2}+L}\|[i]\|[0]^{2^{l/2}}$ and $m_1\| \ldots \| m_{c-1}\| [0]^{2^{l/2}}\|[i]\|[0]^{2^{l/2}+L}$, where $[i] \neq [0]$ and $[i]$s are distinct among pairs.
3. If there is no tag pair that collides, increment the index counter $c \leftarrow c+1$ and if $c \leq s_1 + 1$ then go to step 2, otherwise terminate this phase. If there is (at least) one tag pair that collides, then just execute the following steps.
   (a) Set two integer variables $z_1 = 0$ and $z_2 = 2^{l/2}$.
   (b) Set $z = (z_1 + z_2)/2$. Query to the MAC oracle and receive the corresponding tag pairs of $\log(l)$ distinct message pairs $m_1\| \ldots \| m_{c-1}\| [0]^{2^z+L}\|[i]\|[0]^{2^{l/2}}$ and $m_1\| \ldots \| m_{c-1}\| [0]^{2^z}\|[i]\|[0]^{2^{l/2}+L}$, where $[i] \neq [0]$ and $[i]$s are distinct among pairs.
   (c) If (at least) one tag pair collides, set $z_2 = z$. Otherwise, set $z_1 = z$.
   (d) If $z_2 \neq z_1 + 1$ holds, go to step 3-b. Otherwise, set the height of x_c as $\lambda(x_c) = z_2$, store z_2 in position c in X and increment the index counter $c \leftarrow c+1$. If $c \leq s_1+1$ then go to step 2, otherwise terminate this phase.

Phase 2 (online). Generate a pair of one-block messages (m, m') for each $x_i \in X$, which is used as a filter in Phase 4. The procedure is detailed as below.

1. For all $x_i \in X$ do the following steps.
   (a) Select $2^{l/2}$ distinct one-block messages, append them to $m_1\| \ldots \|m_{i-1}$, and send these newly formatted messages to the MAC oracle. Find the pairs $m_1\| \ldots \|m_{i-1}\|m$ and $m_1\| \ldots \|m_{i-1}\|m'$ that collides on the output of the MAC.
   (b) For all the found pairs (m, m'), choose another random one-block message m'', and query $m_1\| \ldots \|m_{i-1}\|m\|m''$ and $m_1\| \ldots \|m_{i-1}\|m'\|m''$ to the MAC oracle in order to check if their corresponding tags collide again or not. If none collide, go to step 1-a. Otherwise, store a colliding pair (m, m') as the filter for x_i in X and go to the next x_i in step 1.

Phase 3 (offline). Choose $2^l/s_1$ values with their height in $\mathcal{G}_{f_{[0]}}$, and store them in a set Y (sorted according to the height values). The procedure is detailed as below.

1. Initialize a counter c as 0 and the set Y as empty.
2. Choose a new random value y_1 such that $y_1 \notin Y$, and set the chain counter cc to 1.
3. Compute $y_{cc+1} = f_{[0]}(y_{cc})$
4. Check if y_{cc+1} matches a value y stored in Y. If it does, then set the height $\lambda(y_i)$ of y_i (with $1 \leq i \leq cc$) as $\lambda(y) + cc + 1 - i$ and store the $(y_i, \lambda(y_i))$ pairs (with $1 \leq i \leq cc$) in Y.

5. Check if y_{cc+1} matches a previously computed chain value y_i (with $1 \leq i \leq cc$). If it does, then set the height $\lambda(y_j)$ of all values y_j (with $i \leq j \leq cc$) as 0, and the height $\lambda(y_j)$ of y_j (with $1 \leq j < i-1$) as $i-j$. Store the $(y_j, \lambda(y_j))$ pairs (with $1 \leq j \leq cc$) in Y.
6. If no match was found in step 4 or 5, then increment the chain counter $cc \leftarrow cc+1$ and go to step 3. Otherwise, update the counter c by $c \leftarrow c+cc$ and if $c < 2^l/s_1$ then go to step 2, otherwise terminate this phase.

Phase 4 (offline). Recover one intermediate hash value x_i in set X. The procedure is detailed as below.

1. For all $x_i \in X$ do the following steps.
   (a) Get the height $\lambda(x_i)$ of x_i and its filter pair (m, m') from the set X. Get all the $(y, \lambda(y))$ pairs in set Y such that $\lambda(y) = \lambda(x_i)$.
   (b) For each y, we check if $f(y, m) = f(y, m')$ holds or not. If it holds for a y_j, then output y_j as the value of x_i and terminate this phase. If there is no such a y_j, then go to the next x_i in step 1.

Phase 5 (offline). Find a second preimage for the processing of M_{t_2} as the second message part of $H_{K_{in}}(M_t)$. The block length of M_{t_2} is denoted $s_2 = s - s_1$. The procedure is briefly described below. For the complete algorithm please refer to [11].

1. Compute the intermediate hash values x_{s_1}, x_{s_1+1}, ..., x_s from the value x_i recovered at Phase 4, i.e. $x_{i+1} = f(x_i, m_i)$, ..., $x_s = f(x_{s-1}, m_{s-1})$. Note that it is not necessary to compute until x_s.
2. Build a $[\log(s_2), s_2]$-expandable message starting from x_{s_1} to a value denoted as x. More precisely, for any integer i between $\log(s_2)$ and s_2, there is a message $m'_{s_1} \| m'_{s_1+1} \| \cdots \| m'_{s_1+i}$ from the expandable message such that it has i blocks and links from x_{s_1} to x:
$$x = f(\ldots f(f(x_{s_1}, m'_{s_1}), m'_{s_1+1}), \ldots, m'_{s_1+i}).$$
3. Choose $2^l/(s_2 - \log(s_2))$ random one-block messages m, compute $f(x, m)$, and check if this matches to an element of the intermediate hash values set $\{x_{s_1+\log(s_2)}, x_{s_1+1+\log(s_2)}, \ldots, x_s\}$.
4. If a match to x_i (with $s_1 + \log(s_2) \leq i \leq s$) is found, derive the $(i - s_1)$-block long message $m'_{s_1+1} \| m'_{s_1+2} \| \cdots \| m'_i$ from the expandable message, append the blocks $m_{i+1} \| m_{i+2} \| \cdots \| m_s$ to it to produce M'_{t_2}, namely $M'_{t_2} = m'_{s_1+1} \| m'_{s_1+2} \| \cdots \| m'_i \| m_{i+1} \| m_{i+2} \| \cdots \| m_s$.

Phase 6 (online). Forge a valid tag for the challenge M_t.

1. Query message $M'_t = M_{t_1} \| M'_{t_2}$ to the `MAC` oracle, and receive its tag T.
2. Output (M_t, T) where T is a valid tag for M_t.

5.1 Complexity and success probability analysis

Complexity analysis. We use a single compression function call as complexity unit. We evaluated the complexity of each phase as below.

Phase 1:	$O(s_1 \cdot l \cdot \log(l) \cdot 2^{l/2})$	**Phase 2:**	$s_1^2 \cdot 2^{l/2}$	**Phase 3:**	$2^l/s_1$
Phase 4:	$2^l/s_1$	**Phase 5:**	$2^l/s_2$	**Phase 6:**	s

The overall complexity of our generic universal forgery attack therefore depends on the block length s of the target message M_t:

- For the case $s \leq 2^{l/6}$, the overall complexity is dominated by Phase 3 and Phase 5. So we set $s_1 = s_2 = s/2$, and get the overall complexity of $O(2^l/s)$ computations.
- For the case $2^{l/6} < s \leq 2^{5l/6}$, the overall complexity is dominated by Phase 2. So we set $s_1 = 2^{l/6}$, and get the overall complexity of $O(2^{5l/6})$ computations.
- For the case $s > 2^{5l/6}$, the overall complexity is dominated by Phase 6. So we set $s_1 = 2^{l/6}$, and get the overall complexity of $O(s) = 2^{5l/6}$ computations.

Success probability analysis. First, note that we only need to pay attention to the phases that dominate the complexity, since the other phases can be repeated enough times to approach a success probability of 1. For the case $s \leq 2^{l/6}$, we note that Phase 3 always succeeds with probability 1 and the success probability of Phase 5 is 0.63. For the case $2^{l/6} < s \leq 2^{5l/6}$, the success probability of Phase 2 is approximately 1. For the case $s > 2^{5l/6}$, the success probability of Phase 6 is approximately 1 after previous phases were repeated enough times. Therefore, the overall success probability of our attack tends to 1 when repeating a constant time the corresponding complexity dominating phases.

5.2 Experimental verification

For verification purposes, we have implemented the attack by using `HMAC-SHA-256` on a desktop computer. Due to computational and memory limitations, we shortened the input/output bits of the `SHA-256` compression function to 32 bits. In more details, we input a 32-bit value x to the compression function, and the compression function expands it to 256 bits by prepending `0` bits: $\mathtt{0}^{224} \| x$. Then, the compression function also shortens its outputs by only outputting the 32 LSBs. Particularly for Phase 4, we paid attentions to the average number of pairs left after matching the heights between the elements in X and the elements in Y, since it is essential for the complexity advantages. The experiments results confirmed that the universal forgery attack works with the claimed complexity.

6 Conclusion

In this article, we presented the very first generic universal forgery attack against hash-based `MAC`s, and we reduced the gap between the `HMAC` security proof and the

best known attack for this crucial security property. We leave as an open problem if better attacks can be found to further reduce this gap. Our cryptanalysis method is new and uses the information leaked by the distance of a node from the cycle (its height) in the functional graph of the compression function with a fixed message block. We believe other graph properties, even more complex, might be exploitable and could perhaps further improve the generic complexity of universal forgery attacks against hash-based MACs.

References

1. Mihir Bellare. New Proofs for NMAC and HMAC: Security without Collision-Resistance. In Cynthia Dwork, editor, *CRYPTO*, volume 4117 of *Lecture Notes in Computer Science*, pages 602–619. Springer, 2006.
2. Mihir Bellare, Ran Canetti, and Hugo Krawczyk. Keying Hash Functions for Message Authentication. In Neal Koblitz, editor, *CRYPTO*, volume 1109 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 1996.
3. Scott Contini and Yiqun Lisa Yin. Forgery and Partial Key-Recovery Attacks on HMAC and NMAC Using Hash Collisions. In Xuejia Lai and Kefei Chen, editors, *ASIACRYPT*, volume 4284 of *Lecture Notes in Computer Science*, pages 37–53. Springer, 2006.
4. Hans Dobbertin, Antoon Bosselaers, and Bart Preneel. RIPEMD-160: A Strengthened Version of RIPEMD. In Dieter Gollmann, editor, *FSE*, volume 1039 of *Lecture Notes in Computer Science*, pages 71–82. Springer, 1996.
5. Yevgeniy Dodis, Thomas Ristenpart, John P. Steinberger, and Stefano Tessaro. To Hash or Not to Hash Again? (In)Differentiability Results for H^2 and HMAC. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO*, volume 7417 of *Lecture Notes in Computer Science*, pages 348–366. Springer, 2012.
6. Philippe Flajolet and Andrew M. Odlyzko. Random Mapping Statistics. In Jean-Jacques Quisquater and Joos Vandewalle, editors, *EUROCRYPT*, volume 434 of *Lecture Notes in Computer Science*, pages 329–354. Springer, 1989.
7. Philippe Flajolet and Robert Sedgewick. *Analytic Combinatorics*. Cambridge University Press, 2009.
8. Pierre-Alain Fouque, Gaëtan Leurent, and Phong Q. Nguyen. Full Key-Recovery Attacks on HMAC/NMAC-MD4 and NMAC-MD5. In Alfred Menezes, editor, *CRYPTO*, volume 4622 of *Lecture Notes in Computer Science*, pages 13–30. Springer, 2007.
9. Jian Guo, Yu Sasaki, Lei Wang, and Shuang Wu. Cryptanalysis of HMAC/NMAC-Whirlpool. In *ASIACRYPT*, 2013.
10. Bernard Harris. Probability Distributions Related to Random Mappings. *Ann. Math. Statist*, 31(4):1045–1062, 1960.
11. John Kelsey and Bruce Schneier. Second Preimages on n-Bit Hash Functions for Much Less than 2^n Work. In Ronald Cramer, editor, *EUROCRYPT*, volume 3494 of *Lecture Notes in Computer Science*, pages 474–490. Springer, 2005.
12. Jongsung Kim, Alex Biryukov, Bart Preneel, and Seokhie Hong. On the Security of HMAC and NMAC Based on HAVAL, MD4, MD5, SHA-0 and SHA-1 (Extended Abstract). In Roberto De Prisco and Moti Yung, editors, *SCN*, volume 4116 of *Lecture Notes in Computer Science*, pages 242–256. Springer, 2006.

13. Eunjin Lee, Donghoon Chang, Jongsung Kim, Jaechul Sung, and Seokhie Hong. Second Preimage Attack on 3-Pass HAVAL and Partial Key-Recovery Attacks on HMAC/NMAC-3-Pass HAVAL. In Kaisa Nyberg, editor, *FSE*, volume 5086 of *Lecture Notes in Computer Science*, pages 189–206. Springer, 2008.
14. Gaëtan Leurent, Thomas Peyrin, and Lei Wang. New Generic Attacks Against Hash-based MACs. In *ASIACRYPT*, 2013.
15. Ljuben R Mutafchiev. The limit distribution of the number of nodes in low strata of a random mapping. *Statistics & probability letters*, 7(3):247–251, 1988.
16. Thomas Peyrin, Yu Sasaki, and Lei Wang. Generic Related-Key Attacks for HMAC. In Xiaoyun Wang and Kazue Sako, editors, *ASIACRYPT*, volume 7658 of *Lecture Notes in Computer Science*, pages 580–597. Springer, 2012.
17. Bart Preneel and Paul C. van Oorschot. On the Security of Two MAC Algorithms. In Ueli M. Maurer, editor, *EUROCRYPT*, volume 1070 of *Lecture Notes in Computer Science*, pages 19–32. Springer, 1996.
18. G. V. Proskurin. On the Distribution of the Number of Vertices in Strata of a Random Mapping. *Theory Probab. Appl.*, pages 803–808, 1973.
19. Christian Rechberger and Vincent Rijmen. On authentication with hmac and non-random properties. In *Financial Cryptography*, volume 4886 of *Lecture Notes in Computer Science*, pages 119–133. Springer, 2007.
20. Christian Rechberger and Vincent Rijmen. New Results on NMAC/HMAC when Instantiated with Popular Hash Functions. *J. UCS*, 14(3):347–376, 2008.
21. Ronald L. Rivest. The md5 message-digest algorithm. RFC 1321 (Informational), April 1992.
22. Yu Sasaki and Lei Wang. Improved Single-Key Distinguisher on HMAC-MD5 and Key Recovery Attacks on Sandwich-MAC-MD5. In *Selected Areas in Cryptography*, 2013.
23. Lei Wang, Kazuo Ohta, and Noboru Kunihiro. New Key-Recovery Attacks on HMAC/NMAC-MD4 and NMAC-MD5. In Nigel P. Smart, editor, *EUROCRYPT*, volume 4965 of *Lecture Notes in Computer Science*, pages 237–253. Springer, 2008.
24. Xiaoyun Wang, Hongbo Yu, Wei Wang, Haina Zhang, and Tao Zhan. Cryptanalysis on HMAC/NMAC-MD5 and MD5-MAC. In Antoine Joux, editor, *EUROCRYPT*, volume 5479 of *Lecture Notes in Computer Science*, pages 121–133. Springer, 2009.
25. Kan Yasuda. "Sandwich" Is Indeed Secure: How to Authenticate a Message with Just One Hashing. In Josef Pieprzyk, Hossein Ghodosi, and Ed Dawson, editors, *ACISP*, volume 4586 of *Lecture Notes in Computer Science*, pages 355–369. Springer, 2007.
26. Hongbo Yu and Xiaoyun Wang. Full Key-Recovery Attack on the HMAC/NMAC Based on 3 and 4-Pass HAVAL. In Feng Bao, Hui Li, and Guilin Wang, editors, *ISPEC*, volume 5451 of *Lecture Notes in Computer Science*, pages 285–297. Springer, 2009.