New Generic Attacks Against Hash-based MACs

Gaëtan Leurent[1], Thomas Peyrin[2], and Lei Wang[2]

[1] Université Catholique de Louvain, Belgium
`gaetan.leurent@uclouvain.be`
[2] Nanyang Technological University, Singapore
`thomas.peyrin@gmail.com` `wang.lei@ntu.edu.sg`

Abstract. In this paper we study the security of hash-based MAC algorithms (such as `HMAC` and `NMAC`) above the birthday bound. Up to the birthday bound, `HMAC` and `NMAC` are proven to be secure under reasonable assumptions on the hash function. On the other hand, if an n-bit MAC is built from a hash function with a l-bit state ($l \geq n$), there is a well-known existential forgery attack with complexity $2^{l/2}$. However, the remaining security after $2^{l/2}$ computations is not well understood. In particular it is widely assumed that if the underlying hash function is sound, then a generic universal forgery attack should still require 2^n computations and some distinguishing (*e.g.* distinguishing-H but not distinguishing-R) and state-recovery attacks should still require 2^l (or 2^k if $k < l$) computations.

In this work, we show that above the birthday bound, hash-based MACs offer significantly less security than previously believed. Our main result is a generic distinguishing-H and state-recovery attack against hash-based MACs with a complexity of only $\tilde{O}(2^{l/2})$. In addition, we show a key-recovery attack with complexity $\tilde{O}(2^{3l/4})$ against `HMAC` used with a hash functions with an internal checksum, such as `GOST`. This surprising result shows that the use of a checksum might actually weaken a hash function when used in a MAC. We stress that our attacks are generic, and they are in fact more efficient than some previous attacks proposed on MACs instantiated with concrete hash functions.

We use techniques similar to the cycle-detection technique proposed by Peyrin *et al.* at Asiacrypt 2012 to attack `HMAC` in the related-key model. However, our attacks works in the single-key model for both `HMAC` and `NMAC`, and without restriction on the key size.

Key words: `NMAC`, `HMAC`, hash function, distinguishing-H, key recovery, `GOST`.

1 Introduction

Message Authentication Codes (MACs) are crucial components in many security systems. A MAC is a function that takes a k-bit secret key K and an arbitrarily long message M as inputs, and outputs a fixed-length tag of size n bits. The tag is used to authenticate the message, and will be verified by the receiving party using the same key K. Common MAC algorithms are built from block ciphers (*e.g.* CBC-MAC), from hash functions (*e.g.* HMAC), or from universal hash functions (*e.g.* UMAC). In this paper we study MAC algorithms based on hash functions.

As a cryptographic primitive, a MAC algorithm should meet some security requirements. It should be impossible to recover the secret key except by exhaustive search, and it should be computationally impossible to forge a valid MAC without knowing the secret key, the message being chosen by the attacker (existential forgery) or given as a challenge (universal forgery). In addition, cryptanalysts have also studied security notions based on distinguishing games. Informally, the distinguishing-R game is to distinguish a MAC construction from a random function, while the distinguishing-H game is to distinguish a known MAC construction (*e.g.* `HMAC`) instantiated with a known component (*e.g.* `SHA-1`) under a random key from the same construction instantiated with a random component (*e.g.* `HMAC` with a fixed input length random function).

One of the best known MAC algorithm is `HMAC` [2], designed by Bellare *et al.* in 1996. `HMAC` is now widely standardized (by ANSI, IETF, ISO and NIST), and widely deployed, in particular for banking processes or Internet protocols (*e.g.* SSL, TLS, SSH, IPSec). It is a single-key version of the `NMAC` construction, the latter being built upon a keyed iterative hash function H_K, while `HMAC` uses an unkeyed iterative hash function H:

$$\texttt{NMAC}(K_{out}, K_{in}, M) = H_{K_{out}}(H_{K_{in}}(M))$$
$$\texttt{HMAC}(K, M) = H(K \oplus \texttt{opad} \parallel H(K \oplus \texttt{ipad} \parallel M))$$

where `opad` and `ipad` are predetermined constants, and where K_{in} denotes the inner key and K_{out} the outer one.

More generally, a MAC algorithm based on a hash function uses the key at the beginning and/or at the end of the computation, and updates an l-bit state with a compression function. The security of MAC algorithms is an important topic, and both positive and negative results are known. On the one hand, there is a generic attack with complexity $2^{l/2}$ based on internal collisions and length extension [19]. This gives an existential forgery attack, and a distinguishing-H attack. One the other hand, we have security proofs for several MAC algorithms such as HMAC and sandwich-MAC [2,1,29]. Roughly speaking, the proofs show that some MAC algorithms are secure up to the birthday bound ($2^{l/2}$) under various assumptions on the compression function and hash function.

Thanks to those results, one may consider that the security of hash-based MAC algorithms is well understood. However, there is still a strong interest in the security above the birthday bound. In particular, it is very common to expect security 2^k for key recovery attacks if the hash function is sound; the Encyclopedia of Cryptography and Security states explicitly [18] "A generic key recovery attack requires $2^{n/2}$ known text-MAC pairs and 2^{n+1} time" (assuming $n = l = k$). Indeed, key recovery attacks against `HMAC` with a concrete hash function with complexity between $2^{l/2}$ and 2^l have been considered as important results [9,27,30]. Similarly, the best known dinstinguishing-H and state-recovery attacks have a complexity of 2^l (or 2^k if $k < l$), and distinguishing-H attacks on `HMAC` with a concrete hash function with complexity between $2^{l/2}$ and 2^l have been considered as important results [13,20,28].

At the same time with this paper, Naito *et al.* also published a generic distinguishing-H attack on `HMAC`/`NMAC` with a complexity around $O(2^l/l)$ [15]. Note that their attacks are based on the multicollision technique, which is different from our attacks. So we omit the description of their attacks, and refer to [15] for interested readers.

Our contributions. In this paper we revisit the security of hash-based MAC above the birthday bound. We describe a generic distinguishing-H attack in the single key model and with complexity of only $O(2^{l/2})$ computations, thus putting an end to the long time belief of the cryptography community that the best generic distinguishing-H on `NMAC` and `HMAC` requires $\Omega(2^l)$ operations. Instead, we show that a distinguishing-H attack is not harder than a distinguishing-R attack. Our results actually invalidate some of the recently published cryptanalysis works on `HMAC` when instantiated with real hash functions [13,20,28]

Our method is based on a cycle length detection, like the work of Peyrin *et. al* [16], but our utilization is quite different and much less restrictive: instead of iterating many times `HMAC` with small messages and a special related-key, we will use only a few iterations with very long messages composed of almost the same message blocks to observe the cycle and deduct information from it. Overall, unlike in [16], our technique works in the single-key model, for both `HMAC` and `NMAC`, and can be applied for any key size. In addition, leveraging our new ideas, we provide a single-key internal state recovery for any hash-based MAC with only $O(l \cdot \log(l) \cdot 2^{l/2})$ computations.

Finally, this internal state recovery can be transformed into a single-key key recovery attack on `HMAC` with complexity $O(l \cdot 2^{3l/4})$ when instantiated with a hash function using a checksum, such as the `GOST` hash function [6]. A surprising corollary to our results is that incorporating a checksum to a hash function seems to actually reduce the security of the overall `HMAC` design.

We give an overview of our results, and a comparison with some previous analysis in Table 1.

The description of `HMAC`/`NMAC` algorithms and their security are given in Section 2 and we recall in Section 3 the cycle-detection ideas from [16]. Then, we provide in Section 4 the generic distinguishing-H attack. We give in Sections 5 and 6 two different internal state recovery methods, and finally describe our results when the hash function incorporates a checksum in Section 7.

2 Hash-based MAC algorithms

In this paper, we study a category of MAC algorithms based on hash functions, where the key is used at the beginning and at the end of the computation, as described in Figure 1. More precisely, we consider algorithms where: the message processing is done by updating an internal state x using a compression function h; the state is initialized with a key dependent value I_k; and the tag is computed from the last state x_p and the key K by an output function g.

$$x_0 = I_K \qquad x_{i+1} = h(x_i, m_i) \qquad \texttt{MAC}_K(M) = g(K, x_p, |M|)$$

In particular, this description covers `NMAC`/`HMAC` [2], envelope-MAC [24], and sandwich-MAC [29]. The results described in Sections 4 to 6 can be applied to any hash-based MAC, but we focus on `HMAC` for our explanations because it is the most widely used hash-based MAC, and its security has been widely analyzed already. On the other hand the result of Section 7 is specific to MAC algorithms that processes the key as part of the message, such as `HMAC`.

2.1 Description of `NMAC` and `HMAC`

A hash function H is a function that takes an arbitrary length input message M and outputs a fixed hash value of size n bits.

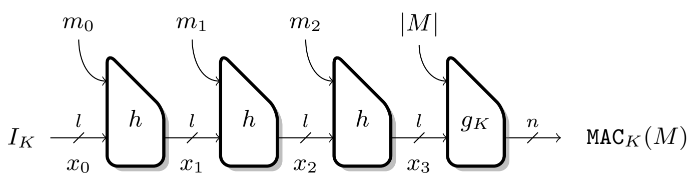


Fig. 1. Hash-based MAC. Only the initial value and the final transformation are keyed.

Virtually every hash function in use today follows an iterated structure like the classical Merkle-Damgård construction [14,4]. Iterative hash functions are built upon successive applications of a so-called compression function h, that takes a b-bit message block and a l-bit chaining value as inputs and outputs a l-bit value (where $l \geq n$). An output function can be included to derive the n-bit hash output from the last chaining variable and from the message length. When $l > n$ (resp. when $l = n$) we say that the hash function is wide pipe (resp. narrow pipe).

The message M is first padded and then divided into blocks m_i of b bits each. Then, the message blocks are successively used to update the l-bit internal state x_i with the compression function h. Once all the message blocks have been processed, the output function g is applied to the last internal state value x_p.

$$x_0 = IV \qquad x_{i+1} = h(x_i, m_i) \qquad \mathsf{hash} = g(x_p, |M|)$$

The MAC algorithm `NMAC` **[2]** uses two l-bit keys K_{out} and K_{in}. `NMAC` replaces the public IV of a hash function H by a secret key K to produce a keyed hash function $H_K(M)$. `NMAC` is then defined as:

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

The MAC algorithm `HMAC` **[2]** is a single-key version of `NMAC`, with $K_{out} = h(IV, K \oplus \mathtt{opad})$ and $K_{in} = h(IV, K \oplus \mathtt{ipad})$ and where `opad` and `ipad` are b-bit constants. However, a very interesting property of `HMAC` for practical utilization is that it can use any key size and can be instantiated with an unkeyed hash function (like `MD5`, `SHA-1`, etc. which have a fixed IV):

$$\mathtt{HMAC}(K, M) = H(K \oplus \mathtt{opad} \parallel H(K \oplus \mathtt{ipad} \parallel M)).$$

where $\parallel$ denotes the concatenation operation. For simplicity of the description and without loss of generality concerning our attacks, in the rest of this article we assume that the key can fit in one compression function message block, i.e. $k \leq b$ (note that K is actually padded to b bits if $k < b$). A graphical description of `HMAC` is given by Figure 2.

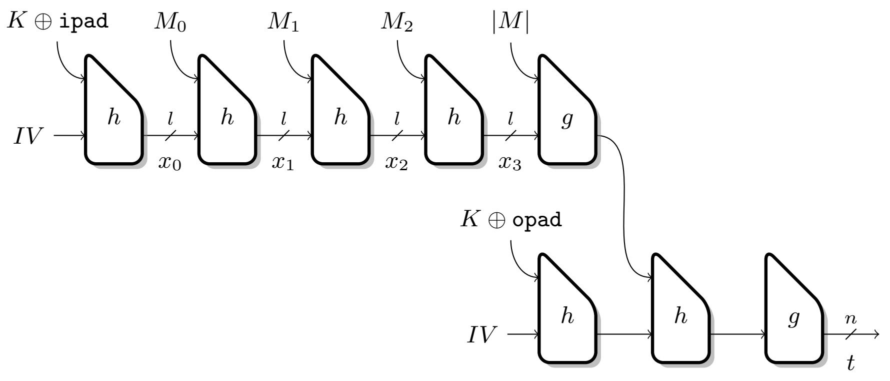


Fig. 2. `HMAC` with a Merkle-Damgård hash function with compression function h, and output function g.

2.2 Security of `NMAC` and `HMAC`

In [1], Bellare proved that the `NMAC` construction is a pseudo-random function (PRF) under the sole assumption that the internal compression function h (keyed by the chaining variable input) is a PRF. The result can be transposed to `HMAC` as well under the extra assumption that h is also a PRF when keyed by the message input.

Concerning key recovery attacks, an adversary should not be able to recover the key in less than 2^k computations for both HMAC and NMAC. In the case of NMAC one can attack the two keys K_{out} and K_{in} independently by first finding an internal collision and then using this colliding pair information to brute force the first key K_{in}, and finally the second one K_{out}.

Universal or existential forgery attacks should cost 2^n computations for a perfectly secure n-bit MAC. However, the iterated nature of the hash functions used inside NMAC or HMAC allows a simple existential forgery attack requiring only $2^{l/2}$ computations [19]. Indeed, with $2^{l/2}$ queries, an adversary can first find an internal collision between two messages (M, M') of the same length during the first hash call. Then, any extra block m added to both of these two messages will lead again to an internal collision. Thus, the attacker can simply forge a valid MAC by only querying for $M \parallel m$ and deducing that $M' \parallel m$ will have the same MAC value.

Concerning distinguishers on HMAC or NMAC, two types have been discussed in the literature: Distinguishing-R and Distinguishing-H attacks, which we define below.

Distinguishing-R. Let $\mathcal{F}_n$ be the set of n-bit output functions. We denote F_K the oracle on which the adversary $\mathcal{A}$ can make queries. The oracle is instantiated either with $F_K = \mathtt{HMAC}_K$ (with K being a randomly chosen k-bit key) or with a randomly chosen function R_K from $\mathcal{F}_n$. The goal of the adversary is to distinguish between the two cases and its advantage is given by

$$Adv(\mathcal{A}) = |\Pr[\mathcal{A}(\mathtt{HMAC}_K) = 1] - \Pr[\mathcal{A}(R_K) = 1]| .$$

Obviously the collision-based forgery attack detailed above gives directly a distinguishing-R attack on NMAC and HMAC. Thus, the expected security of HMAC and NMAC against distinguishing-R attacks is $2^{l/2}$ computations.

Distinguishing-H. The attacker is given access to an oracle $\mathtt{HMAC}_K$ and the compression function of the HMAC oracle is instantiated either with a known dedicated compression function h or with a random chosen function r from $\mathcal{F}_l^{b+l}$ (the set of $(b+l)$-bit to l-bit functions), which we denote $\mathtt{HMAC}_K^h$ and $\mathtt{HMAC}_K^r$ respectively. The goal of the adversary is to distinguish between the two cases and its advantage is given by

$$Adv(\mathcal{A}) = \left|\Pr[\mathcal{A}(\mathtt{HMAC}_K^h) = 1] - \Pr[\mathcal{A}(\mathtt{HMAC}_K^r) = 1]\right| .$$

The distinguishing-H notion was introduced by Kim *et al.* [13] for situations where the attacker wants to check which cryptographic hash function is embedded in HMAC. To the best of our knowledge, the best known generic distinguishing-H attack requires 2^l computations.

Related-key attacks. At Asiacrypt 2012 [16], a new type of generic distinguishing (distinguishing-R or distinguishing-H) and forgery attacks for HMAC was proposed. These attacks are in the related-key model and can apply even to wide-pipe proposals, but they only work for HMAC, and only when a special restrictive criterion is verified: the attacker must be able to force a specific difference between the inner and the outer keys (with the predefined values of opad and ipad in HMAC, this criterion is verified when $k = b$). The idea is to compare the cycle length when iterating the HMAC construction on small messages with a key K, and the cycle length when iterating with a key $K' = K \oplus \mathtt{opad} \oplus \mathtt{ipad}$.

Attacks on instantiations with concrete hash functions. Because of its widespread use in many security applications, HMAC has also been carefully scrutinized when instantiated with a concrete hash function, exploiting weaknesses of some existing hash function. In parallel to the recent impressive advances on hash function cryptanalysis, the community analyzed the possible impact on the security of HMAC when instantiated with standards such as MD4 [21], MD5 [22], SHA-1 [25] or HAVAL. In particular, key-recovery attacks have been found on HMAC-MD4 [9,27], HMAC-HAVAL [30] and HMAC-Whirlpool [11]. Concerning the distinguishing-H notion, one can cite for example the works from Kim *et al.* [13], Rechberger *et al.* [20], Wang *et al.* [28], and Sasaki and Wang [23].

However, to put these attacks in perspective, it is important to know the complexity of *generic* attacks, that work even with a good hash function.

3 Cycle detection for HMAC

Our new attacks are based on some well-known properties of random functions.

3.1 Random mapping properties on a finite set

Let us consider a random function f mapping n bits to n bits and we denote $N = 2^n$. We would like to know the structure of the functional graph defined by the successive iteration of this function, for example the expected number of components, cycles, etc. First, it is easy to see that each component will contain a single cycle with several trees linked to it. This has already been studied for a long time, and in particular we recall two theorems from Flajolet and Odlyzko [7].

Theorem 1 ([7, Th. 2]). *The expectations of the number of components, number of cyclic points, number of terminal points, number of image points, and number of k-th iterate image points 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 points: $(1 - e^{-1})N$
(v) # k-th iterate image points: $(1 - \tau_k)N$
where τ_k *satisfies* $\tau_0 = 0$, $\tau_{k+1} = e^{-1+\tau_k}$.

In particular, a random mapping has only a logarithmic number of distinct components, and the number of cyclic points follows the square root of N.

By choosing a random starting point P and iterating the function f, one will follow a path in the functional graph starting from P, that will eventually connect to the cycle of the component in which P belongs, and we call **tail length** the number of points in this path. Similarly, we call **cycle length** the number of nodes in the cycle. Finally, the number of points in the non-repeating trajectory from P is called the **rho length**, and we call **α-node** of the path the node that connects the tail and the cycle.

Theorem 2 ([7, Th. 3]). *Seen from a random point 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: $\sqrt{\pi N/8}$

One can see that, surprisingly, in a random mapping most of the points tend to be grouped together in a single giant component, and there is a giant tree with a significant proportion of the points. The asymptotic expectation of the maximal features is given by Flajolet and Sedgewick [8].

Theorem 3 ([8, VII.14]). *In a random mapping of size N, the largest tree has an expected size of $\delta_1 N$ with $\delta_1 \approx 0.48$ and the largest component has an expected size of $\delta_2 N$ with $\delta_2 \approx 0.7582$.*

These statistical properties will be useful to understand the advantage of our attacks. We show the functional graph of a simple random-looking function in Figure 3.

3.2 Using cycle-detection to obtain some secret information

In this article and as in [16], we will study the functional graph structure of a function to derive a distinguisher or obtain some secret information. More precisely, in [16] Peyrin *et al.* observed that the functional graph structure of `HMAC` was the same when instantiated with a key K or with a related key $K' = K \oplus \texttt{ipad} \oplus \texttt{opad}$ (note that in order to be able to query this related-key K', the key K has to be of size b or $b-1$, which is quite restrictive). This is a property that should not exist for a randomly chosen function and they were able to detect this cycle structure by measuring the cycle length in both cases K and K', and therefore obtaining a distinguishing-R attack for `HMAC` in the related-key model. In practice, the attacker can build and observe the functional graph of `HMAC` by simply successively querying the previous n-bit output as new message input.

In this work, instead of studying the structure of the functional graph of `HMAC` directly, we will instead study the functional graph of the internal compression function h with a fixed message block: we denote $h_M(X) = h(X, M)$. We aim to obtain some information on h_M that we can propagate outside the `HMAC` structure. This is therefore perfectly suited for a distinguishing-H attack, which requires the attacker to exhibit a property of h when embedded inside the `HMAC` construction. We can traverse the functional graph of h_M by querying the `HMAC` oracle with a long message composed of many repetitions of the fixed message block M. The issue is now to detect some properties of the functional graph of h_M inside `HMAC` and without knowing the secret key. We explain how to do that in the next section.

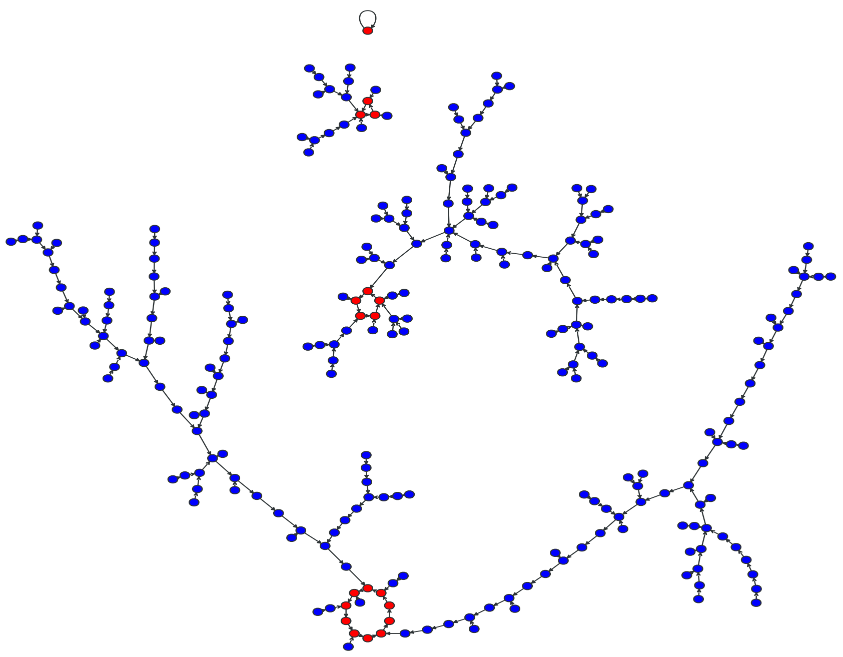


Fig. 3. Functional graph of Keccak (`SHA-3`) with 8-bit input and 8-bit output.

4 Distinguishing-H attack for hash-based MACs

In the rest of the article, we use the notation $[x]^k$ to represent the successive concatenation of k message blocks x, with $[x] = [x]^1$.

4.1 General description

In order to derive a distinguishing-H attack, we need to do some offline computations with the target compression function h and use this information to compare online with the function embedded in the MAC oracle. We use the structure of the functional graph of $h_{[0]}$ to derive our attack (of course we can choose any fixed message block). We can travel in the graph by querying the oracle using consecutive [0] message blocks. However, since the key is unknown, we do not know where we start or where we stop in this graph. We have seen in the previous section that the functional graph of a random function is likely to have a giant component containing most of the nodes. We found that the cycle size of the giant component of $h_{[0]}$ is a property that can be efficiently tested.

More precisely, we first compute the cycle size of the giant component of $h_{[0]}$ offline; we denote it as L. Then, we measure the cycle size of the giant component of the unknown function by querying the MAC oracle with long messages composed of many consecutive [0] message blocks. If no length padding is used in the hash function, this is quite simple: we just compare $\texttt{MAC}([0]^{2^{l/2}})$ and $\texttt{MAC}([0]^{2^{l/2}+L})$. With a good probability, the sequence of $2^{l/2}$ zero block is sufficiently long to enter the cycle, and if the cycle has length L, the two MAC outputs will collide.

Unfortunately, this method does not work because the lengths of the messages are different, and thus the last message block with the length padding will be different and prevent the cycle collision to propagate to the MAC output. We will use a trick to overcome this issue, even though the basic method remains the same. The idea isto build a message M going twice inside the cycle of the giant component, so that we can add L [0] message blocks in the first cycle to obtain a message M_1 and L [0] message blocks in the second cycle to obtain M_2. This is depicted in Figure 4: M_1 will cycle L [0] message blocks in the first cycle (red dotted arrows), while M_2 will cycle L [0] message blocks in the second cycle (blue dashed arrows), and thus they will both have the same length overall.

To perform the distinguishing-H attack, the adversary simply randomly selects an initial message block m and query $M_1 = m \parallel [0]^{2^{l/2}} \parallel [1] \parallel [0]^{2^{l/2}+L}$ and $M_2 = m \parallel [0]^{2^{l/2}+L} \parallel [1] \parallel [0]^{2^{l/2}}$ to the MAC oracle, and deduce that the target function is used to instantiate the oracle if the two MAC values are colliding. The [1] message

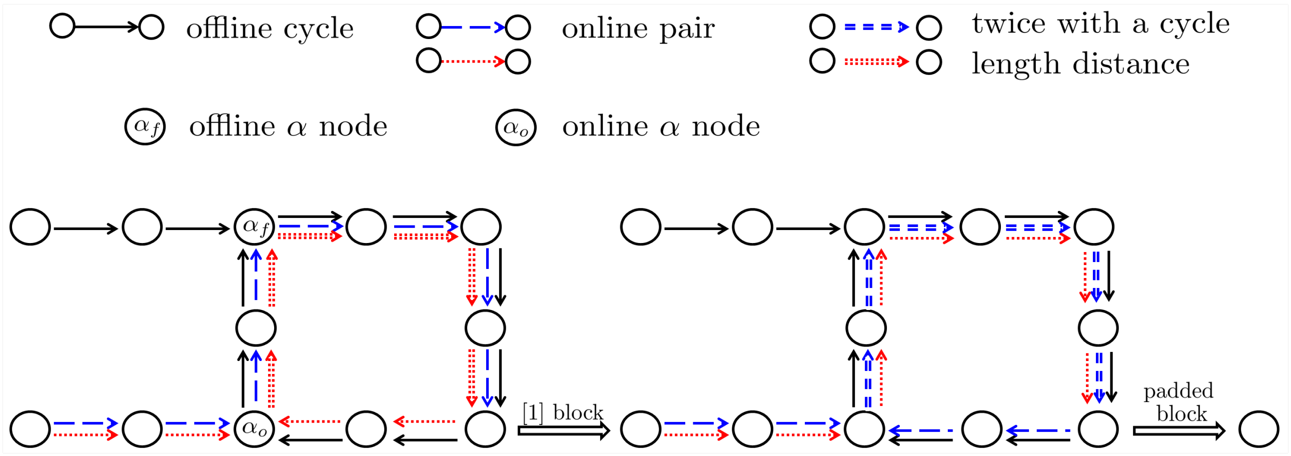


Fig. 4. Distinguishing-H attack

block is used to quit the cycle and randomize the entry point to return again in the giant component. We give below a detailed attack procedure and complexity analysis.

This attack is very interesting as the first generic distinguishing-H attack on `HMAC` and `NMAC` with a complexity lower than 2^l. However, we note that the very long message length might be a limitation. In theory this is of no importance and our attack is indeed valid, but in practice some hash functions forbid message inputs longer than a certain length. To address this issue we provide an alternative attack in Section 6 using shorter messages, at the cost of a higher complexity.

4.2 Detailed attack process

1. (offline) Search for a cycle in the functional graph of $h_{[0]}$ and denote L its length.
2. (online) Choose a random message block m and query the `HMAC` value of the two messages $M_1 = m \,\|\, [0]^{2^{l/2}} \,\|\, [1] \,\|\, [0]^{2^{l/2}+L}$ and $M_2 = m \,\|\, [0]^{2^{l/2}+L} \,\|\, [1] \,\|\, [0]^{2^{l/2}}$.
3. If the `HMAC` values of M_1 and M_2 collide then output 1, otherwise output 0.

4.3 Complexity and success probability analysis

We would like to evaluate the complexity of the attack. The first step will require about $2^{l/2}$ offline computations to find the cycle for the target compression function h. It is important to note that we can run this first step several times in order to ensure that we are using the cycle from the largest component of the functional graph of $h_{[0]}$. The second step makes two queries of about $2^{l/2} + 2^{l/2} + L \simeq 3 \cdot 2^{l/2}$ message blocks each. Therefore, the overall complexity of the attack is about $2^{l/2+3}$ compression function computations.

Next, we evaluate the advantage of the adversary in winning the distinguishing-H game defined in Section 2.2. We start with the case where the oracle is instantiated with the real compression function h. The adversary will output 1 if a collision happens between the `HMAC` computations of M_1 and M_2. Such a collision can happen when the following conditions are satisfied:

- The processing of the random block m sets us in the same component (of the functional graph of $h_{[0]}$) as in the offline computation. Since we ensured that the largest component was found during the offline computation, and since it has an average size of $0.7582 \cdot 2^l$ elements according to Theorem 2, this event will happen with probability 0.7582.
- The $2^{l/2}$ [0] message blocks concatenated after m are enough to reach the cycle of the component, i.e. the tail length when reaching the first cycle is smaller than $2^{l/2}$. Since the average tail length is less than $2^{l/2}$ elements, this will happen with probability more than 1/2.
- The processing of the block [1] sets us in the same component (of the functional graph of $h_{[0]}$) as in the offline computation; again the probability is 0.7582.
- The $2^{l/2}$ [0] message blocks concatenated after [1] are enough to reach the cycle of the component; again this happens with probability 1/2.

The collision probability is then $(0.7582)^2 \times 1/4 \simeq 0.14$ and thus we have that $\Pr[\mathcal{A}(\mathtt{HMAC}_K^h) = 1] \geq 0.14$. In the case where the oracle is instantiated with a random function r, the adversary will output 1 if and only if a random `HMAC` collision happens between messages M_1 and M_2. Such a collision can be obtained if the processing of any of the last $[0]^{2^{l/2}}$ blocks of the two messages leads to an internal collision, therefore with negligible probability $2^{l/2} \cdot 2^{-l} = 2^{-l/2}$. Overall, the adversary advantage is equal to $Adv(\mathcal{A}) = \left|0.14 - 2^{-l/2}\right| \simeq 0.14$.

5 Internal state recovery for NMAC and HMAC

This section presents an internal-state recovery attack that extends the distinguishing-H attack.

5.1 General description

In order to extent the distinguishing-H attack to a state recovery attack, we observe that there is a high probability that the α-node reached in the online phase is the root of the giant tree of the functional graph of $h_{[0]}$. More precisely, we can locate the largest tree and the corresponding α-node in the offline phase, by repeating the cycle search a few times. We note that $\delta_1 + \delta_2 > 1$, therefore the largest tree is in the largest component with asymptotic probability one (see Theorem 3). Thus, assuming that the online phase succeeds, the α-node reached in the online phase is the root of the largest tree with asymptotic probability $\delta_1/\delta_2 \approx 0.63$. If we can locate the α-node of the online phase (i.e. we deduce its block index inside the queried message), we immediately get its corresponding internal state value from the offline computations.

Since the average rho length is $\sqrt{\pi 2^l/2}$ we can not use a brute-force search to locate the α-node, but we can use a binary search instead. We denote the length of the cycle of the giant component as L, and the follow the distinguishing-H attack to reach the cycle. We choose a random message block m and we query the two messages $M_1 = m \,\|\, [0]^{2^{l/2}} \,\|\, [1] \,\|\, [0]^{2^{l/2}+L}$ and $M_2 = m \,\|\, [0]^{2^{l/2}+L} \,\|\, [1] \,\|\, [0]^{2^{l/2}}$. If the MAC collide, we know that the state reached after processing $m \,\|\, [0]^{2^{l/2}}$ is located inside the main cycle. We use a binary search to find the smallest X so that the state reached after $m \,\|\, [0]^X$ is in the cycle.

The first step of the binary search should decide whether the node reached after $m \,\|\, [0]^{2^{l/2-1}}$ is also inside the first online cycle or not. More precisely, we check if the two messages $M'_1 = m \,\|\, [0]^{2^{l/2-1}} \,\|\, [1] \,\|\, [0]^{2^{l/2}+L}$ and $M'_2 = m \| [0]^{2^{l/2-1}+L} \| [1] \| [0]^{2^{l/2}}$ also give a colliding tag. If it is the case, then the node after processing $m \| 2^{l/2-1}$ is surely inside the cycle, which implies that the α-node is necessarily located in the first $2^{l/2-1}$ zero blocks. On the other hand, if the tags are not colliding, we cannot directly conclude that the α-node is located in the second half since there is a non-negligible probability that the α-node is in the first half but the $[1]$ block is directing the paths in M'_1 and M'_2 to distinct components in the functional graph of $h_{[0]}$ (in which case the tag are very likely to differ). Therefore, we have to test for collisions with several couples $M^k_1 = m \,\|\, [0]^{2^{l/2-1}} \,\|\, [k] \,\|\, [0]^{2^{l/2}+L}$, $M^k_2 = m \,\|\, [0]^{2^{l/2-1}+L} \,\|\, [k] \,\|\, [0]^{2^{l/2}}$, and if none of them collide we can safely deduce that the α-node is located in the second $2^{l/2-1}$ zero blocks. Overall, one such step reduces the number of the candidate nodes by a half and we simply continue this binary search in order to eventually obtain the position of the α-node with $\log_2(2^{l/2}) = l/2$ iterations.

5.2 Detailed attack process

1. (offline) Search for a cycle in the functional graph of $h_{[0]}$ and denote L its length.
2. (online) Find a message block m such that querying the two messages $M_1 = m \,\|\, [0]^{2^{l/2}} \,\|\, [1] \,\|\, [0]^{2^{l/2}+L}$ and $M_2 = m \,\|\, [0]^{2^{l/2}+L} \,\|\, [1] \,\|\, [0]^{2^{l/2}}$ leads to the same HMAC output. Let X_1 and X_2 be two integer variables, initialized to the values 0 and $2^{l/2}$ respectively.
3. (online) Let $X' = (X_1 + X_2)/2$. Select $\beta \log(l)$ distinct message blocks $[k]$, and for each of them query the HMAC output for messages $M'_1 = m \,\|\, [0]^{X'} \,\|\, [k] \,\|\, [0]^{2^{l/2}+L}$ and $M'_2 = m \,\|\, [0]^{X'+L} \,\|\, [k] \,\|\, [0]^{2^{l/2}}$. If at least one of the (M'_1, M'_2) pairs leads to a colliding HMAC output, then set $X_2 = X'$. Otherwise, set $X_1 = X'$. We use $\beta = 4.5$ as explained later.
4. (online) If $X_1 + 1 = X_2$ holds, output X_2 as the block index of the α-node. Otherwise, go back to the previous step.

5.3 Complexity and success probability analysis

Complexity. We would like to evaluate the complexity of the attack. The first step will require about $2^{l/2}$ offline computations to find the cycle. Again, it is important to note that we can run this first step several times in order to ensure that we are using the cycle from the biggest component of the functional graph of $h_{[0]}$. The second step repeats the execution of the distinguishing-H attack from Section 4, which requires $6 \cdot 2^{l/2}$ computations for a success probability of 0.14, until it finds a succeeding message block m. Therefore, after trying a few m values, we have probability very close to 1 to find a valid one. The third and fourth steps will be executed about $l/2$ times (for the binary search), and the third steps performs $2 \cdot \beta \log(l)$ queries of about $2^{l/2} + 2^{l/2} + L \simeq 3 \cdot 2^{l/2}$ message blocks each. Therefore, the overall complexity of the attack is about $3\beta \cdot l \cdot \log(l) \cdot 2^{l/2}$ compression function computations.

Success probability. Next we evaluate the success probability that the attacker recovers the internal state and this depends on the success probability of the binary search steps. We start with the case where the node after $m \| [0]^{X'}$ is inside the first online cycle. The third step will succeed as long as at least one of the (M_1', M_2') pairs collide on the output (we can omit the false positive collisions which happen with negligible probability). One pair (M_1', M_2') will indeed collide if:

- The random block $[k]$ sends both messages to the main component of the functional graph of $h_{[0]}$. Since it has an expected size of $\delta_2 \cdot 2^l$ (see Theorem 3), this is the case with probability δ_2^2.
- The $2^{l/2}$ $[0]$ message blocks concatenated after $[k]$ are enough to reach the cycle, i.e. the tail length when reaching the second cycle is smaller than $2^{l/2}$. Since the average tail length is smaller than $2^{l/2}$ elements, this will happen with probability 1/2 for each message.

After trying $\beta \log(l)$ pairs, the probability that at least one pair collides is $1 - (1 - \delta_2^2/4)^{\beta \log(l)}$. If we use $\beta = -1/\log(1-\delta_2^2/4) \approx 4.5$, this gives a probability of $1 - 1/l$. On the other hand, if the node after $m \| [0]^{X'}$ is not inside the cycle, the third step will succeed when no random collision occurs among the $\beta \log(l)$ tries, and such collisions happen with negligible probability. Overall, since there are $l/2$ steps in the binary search, the average success probability of the binary search is $(1 - 1/l)^{l/2} \geq e^{-1/2} \approx 0.6$.

Finally, the attack succeeds if the α-node is actually the root of the giant tree, as computed in the offline phase. This is the case with probability δ_1/δ_2, and the success probability of the full state recovery attack is $\delta_1/\delta_2 \cdot e^{-1/2} \approx 0.38$.

5.4 Transformation from a long message to a short message

Note that the attack procedure in this section recovers some internal state of a message, which is up to $O(2^{l/2})$ blocks long. The long length of the message may become a problem in applications of our internal state recovery attack, e.g., it may cause the complexity becomes too high. Thus it is interesting and important to mention that the internal state of a long message denoted as M can be utilized to obtain an internal state of a extremely short message in a straightforward way. More precisely, we append random one-block messages m to M, and query $M \| m$ to `MAC`. Meanwhile, we also query one-block messages to `MAC`. Then we filter out a collision on the inner hash function between a long message and a short message $H_{K_{in}}(M \| m) = H_{K_{in}}(m')$. Finally by computing the value of $H_{K_{in}}(M \| m)$ using the previously recovered internal state of M, we get an internal state of m', which is one block long. A more detailed description and procedure is referred to Section 7.

6 Internal state recovery for `NMAC` and `HMAC` with shorter messages

In this section, we describe an alternative internal-state recovery attack using shorter messages, that can also be used as a distinguishing-H attack with shorter messages. The attacks of sections 4 and 5 have a complexity of $O(2^{l/2})$ using a small number of messages of length $2^{l/2}$; in this section we describe an attack with complexity $O(2^{3l/4})$ using $2^{l/2}$ messages of length $2^{l/4}$. More generally, if the message size is limited to 2^s blocks ($s \leq l/4$), then the attack requires $2^{3l/4-s}$ messages, and has a complexity of $O(2^{l-s})$.

6.1 Entropy loss in collisions finding

While the previous attacks are based on detecting cycles in the graph of a random function, this attack is based on the fact that finding collisions by iterating a random function does not give a random collision: some particular collisions are much more likely to be found than others. This fact is well known in the context of collision search; for instance van Oorschot and Wiener [26] recommend to randomize the function regularly when looking for a large number of collisions. In this attack, we exploit this property to our advantage: first we use a collision finding algorithm to locate collisions in h_M with a fixed M; then we query the MAC oracle with messages with long repetitions of the block M and we detect collisions in the internal state; since the collisions found in this way are not randomly distributed, there is a good probability that we will reach one the collisions that was previously detected in the offline phase.

More precisely, we consider the collision finding algorithm using fixed length chains described in Algorithm 1. This algorithm computes $f^{2^s}(x)$ for many different x's, and when a value is reached twice ($f^{2^s}(x) = f^{2^s}(x')$), it restarts the chain to locate the exact point where they join.

Intuitively, if $s = 0$, a collision can only be detected when two preimages of the point are taken as starting point. If we disregard multi-collisions in f (*i.e.* points with more than 2 preimages), the algorithm will find a collision after $2^{l/2}$ iterations, and each collision is equally likely.

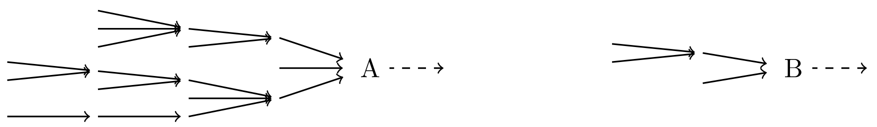
On the other hand, with $s = l/2$, most of the 2^s-th images are cyclic points, and we expect a match after about $2^{l/4}$ iterations (*i.e.* $2^{3l/4}$ evaluations of f) since there are about $2^{l/2}$ cyclic points (see Theorem 1). Moreover, when a collision $f^{2^s}(x) = f^{2^s}(x')$ is found, there is a high probability that x is in the giant tree,

Algorithm 1 Collision finding with fixed length chains (length 2^s)

```
$T \leftarrow \{\}$
$x \leftarrow \mathsf{Rand}()$
**loop**
    $x \leftarrow x + 1$
    $y \leftarrow f^{2^s}(x)$
    **if** $T\{y\} = \varnothing$ **then**
        $T\{y\} \leftarrow x$
    **else**
        $x' \leftarrow T\{y\}$
        **repeat**
            $x \leftarrow f(x)$
            $x' \leftarrow f(x')$
        **until** $x = x'$
        **return** $x$
    **end if**
**end loop**
```

while x' is in the giant component but not in the giant tree. Therefore, with high probability, the collision that we detect will be the α-node of the giant tree; in this case the output of Algorithm 1 has low entropy.

In general, with $0 \le s \le l/2$ every collision in f can be found by Algorithm 1 with a non-zero probability, but the collisions with many iterated preimages are more likely, and the bias in the distribution increases with s. For instance, let us consider two nodes with the following preimages:



Node B can only be detected with a single pair of starting points, but A can be detected with 3 different pairs if $s = 0$, with $3 + 6 = 9$ pairs if $s = 1$, and with $3 + 6 + 9 = 18$ pairs if $s \ge 2$.

To study the entropy loss in the output of Algorithm 1 formally, we use the Rényi entropy, or collision entropy.

Definition 1 (Rényi entropy). *For a discrete random variable X with possible outcome $1, 2, \dots$ and corresponding probabilities $p_i = \Pr(X = i)$, the Rényi entropy, or collision entropy, is defined as:*

$$H_2(X) = -\log_2 \sum_i p_i^2$$

Property 1. If X and Y are independent and identically distributed, the probability of a collision is $P(X = Y) = 2^{-H_2(X)}$.

We can now study the behavior of Algorithm 1. We didn't manage to prove formally the properties we use, but we make reasonable conjectures. First we estimate how many starting points are needed to find collisions:

Conjecture 1. The Rényi entropy of the 2^s-th iteration (with $0 \le s \le l/2$) of a random mapping is about $l - s$ bits.

Corollary 1. *Algorithm 1 take about 2^{l-s} steps before finding a collision.*

The extreme cases $s = 0$ and $s = l/2$ are easy to prove, and we conjecture the formula for intermediate values. This is related to the number of 2^s-th images as given in Theorem 1.

We also make a similar conjecture about the distribution of the output of Algorithm 1:

Conjecture 2. The Rényi entropy of collisions found by iterating chains of length 2^s (with $0 \le s \le l/2$) in a random mapping is about $l - 2s$ bits.

Again, the extreme cases $s = 0$ and $s = l/2$ are easy to analyze (for $s = l/2$ the α-node of the giant tree is found with constant probability, therefore the collision entropy is close to zero), and we conjecture the formula for intermediate values.

To verify the conjectures we performed experiments with random functions for small values of l. With $l \le 32$, we can build the full table of powers of f in memory, and compute exactly the number of 2^s-th preimages of all points (to verify Conjecture 1), and the number of pairs that can be used to detect each collision (to verify Conjecture 2). Figure 5 shows the results of our experiments. We find that Conjecture 1 gives very accurate predictions, while Conjecture 2 is slightly optimistic. We give some additional experimental results in Figure 6.

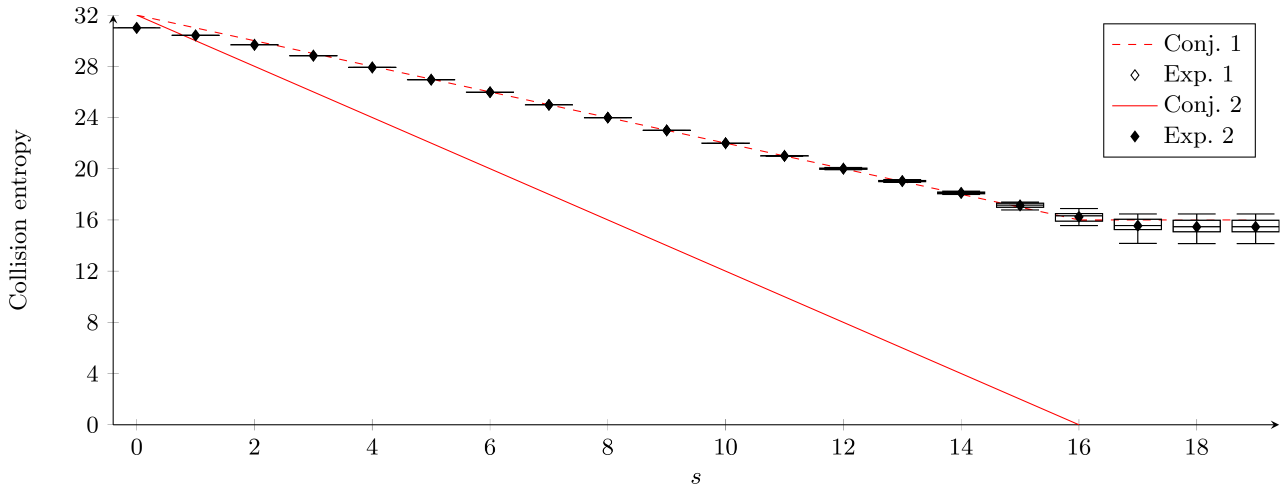


Fig. 5. Experimental verification of conjecture 1 and conjecture 2 with 8 random functions ($l = 32$).

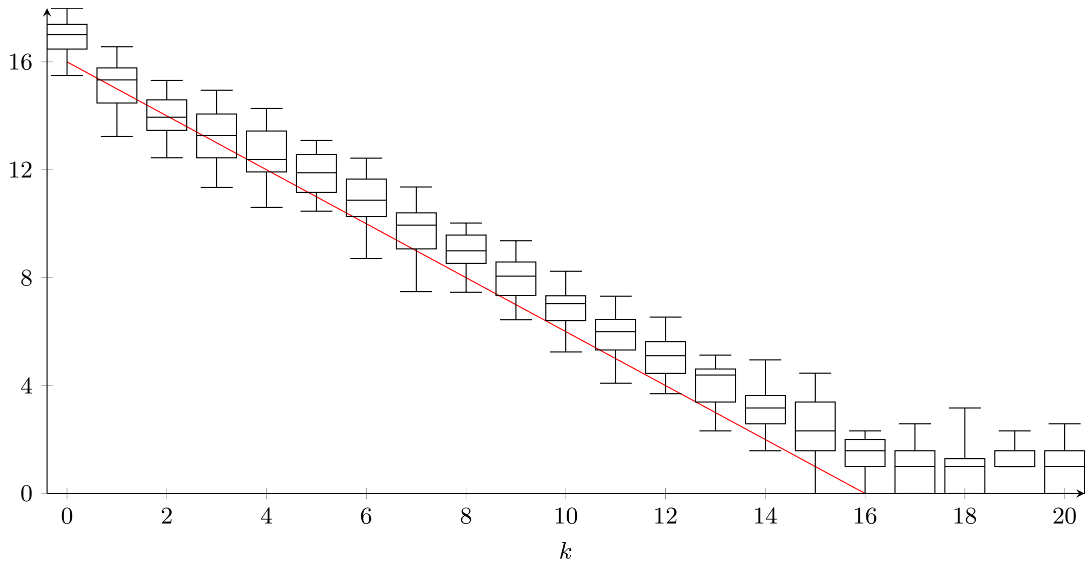


Fig. 6. Experimental verification of conjecture 2. We generate collisions with chains of 2^k iterations, following step 1 of the attack, and we report the number of collision points needed before a point is found twice. We use $l = 32$, and for each k, we ran 32 experiments with random functions; we compare the results with the value $l/2 - k$ that we expect for the conjecture.

6.2 Detailed attack process

This analysis of collision finding leads to our new state recovery algorithm with short messages. The algorithm is based on computing 2^t chains of length 2^s, and exploiting the small entropy of the collisions detected in this way.

1. (offline) Iterate the compression function with a zero message 2^s times, starting from 2^t random starting points. Sort the end points, and locate 2^c collisions.
2. (offline) For each collision, restart the chains, and find the point x where they collide. Then, find a pair of (non-zero) message blocks m_x, m'_x that give a collision starting from this point. This will be used in online step 5 to test candidate points.
3. (online) Query the oracle with 2^t messages $M_i = [i] \parallel [0]^{2^s}$. Sort the tags, and locate 2^c collisions.
4. (online) For each collision $\mathtt{MAC}(M_i) = \mathtt{MAC}(M_j)$, use a binary search to find the smallest κ such that $\mathtt{MAC}([i] \parallel [0]^\kappa) = \mathtt{MAC}([j] \parallel [0]^\kappa)$. Note that a collision for κ implies a collision for any $s \geq \kappa$, and we always consider pairs of messages with the same length, so the padding is not problem. We denote $[i] \parallel [0]^\kappa$ as μ_{ij}
5. (online) For each collision compare $\mathtt{MAC}(\mu_{ij} \parallel m_x)$ and $\mathtt{MAC}(\mu_{ij} \parallel m'_x)$ for all x's detected at step 1. If $\mathtt{MAC}(\mu_{ij} \parallel m_x) = \mathtt{MAC}(\mu_{ij} \parallel m'_x)$, then with very high probability the state reached after processing μ_{ij} is x.

6.3 Complexity analysis

According to Conjecture 1, that the number of collisions detected at step 1 and 3 is $c \approx 2t - (l - s)$. Similarly, Conjecture 2 shows that there will be a match between the set of collisions found in step 1 and the set of collisions found in step 4 if $c \geq l/2 - s$. Therefore the optimal parameters satisfy:

$$c = l/2 - s \qquad\qquad t = 3l/4 - s$$

We can compute the complexity of each step of the attack:

Step 1:	$2^{t+s} = 2^{3l/4}$	**Step 2:**	$2^{c+l/2} = 2^{l-s}$
Step 3:	$2^{t+s} = 2^{3l/4}$	**Step 4:**	$s \cdot 2^{c+s} = s \cdot 2^{l/2}$
Step 5:	$2^{2c+s} = 2^{l-s}$		

Therefore, the total complexity of the attack is $O(2^{3l/4})$ if $s \geq l/4$, and $O(2^{l-s})$ otherwise.

7 Key recovery for HMAC based on a hash function with an internal checksum

In this section we study HMAC used with a hash function with an internal checksum, such as GOST. We show that the checksum does not prevent the distinguishing-H and state recovery attack, but more surprisingly the checksum actually allows to mount a full key-recovery attack significantly more efficient than exhaustive search.

A hash function with an internal checksum computes the sum of all message blocks, and uses this sum as an extra input to the finalization function. The sum can be computed for any group operation, but it will usually be an XOR sum or a modular addition. We use the XOR sum $\mathtt{Sum}^{\oplus}$ to present our attack, but it is applicable with any group operation.

The checksum technique has been used to enhance the security of a hash function, assuming the controlling the checksum would be an additional challenge for an adversary. While previous work argued its effectiveness [10], our result reveals a surprising fact that incorporating a checksum into a hash function could even weaken its security in some applications such as HMAC.

A notable example of a hash function with a checksum is the GOST hash function, which has been standardized by the Russian Government [6] and by IETF [5]. HMAC-GOST has also been standardized by IETF [17] and is implemented in OpenSSL. GOST uses parameters $n = l = b$, and uses a separate call to process the message length, as follows:

$$x_0 = IV \qquad x_{i+1} = h(x_i, m_i) \qquad x_* = h(x_p, |M|)$$
$$\sigma_0 = 0 \qquad \sigma_{i+1} = \sigma_i \oplus m_i \qquad \mathsf{hash} = g(x_*, \sigma_p)$$

If this section we describe the attack on GOST-like functions following this structure; Figure 7 shows an HMAC computation with a GOST-like hash function. We give more general attacks when the output is computed as $g(x_p, |M|, \sigma_p)$ in Appendix A.

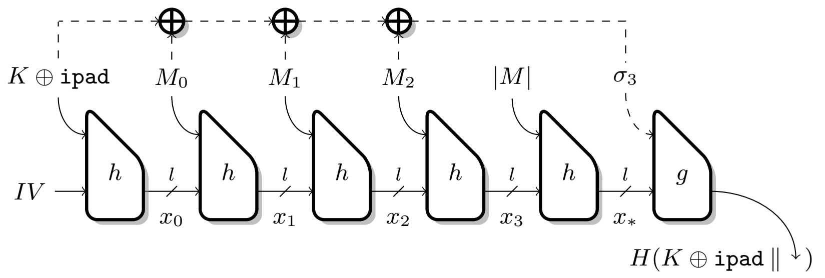


Fig. 7. HMAC based on a hash function with a checksum (dashed lines) and a length-padding block. We only detail the first hash function call.

7.1 General description

In HMAC, $K \oplus \mathtt{ipad}$ is prepended to a message M, and $(K \oplus \mathtt{ipad}) \,\|\, M$ is hashed by the underlying hash function H. Therefore, the final checksum value is $\sigma_p = \mathtt{Sum}^{\oplus}((K \oplus \mathtt{ipad}) \,\|\, M) = K \oplus \mathtt{ipad} \oplus \mathtt{Sum}^{\oplus}(M)$. In this attack, we use the state recovery attack to recover the internal state x_* before the checksum is used and we take advantage of the fact that the value σ_p actually contains the key, but can still be controlled by changing the message.

More precisely, we use Joux's multicollision attack [12] to generate a large set of messages with the same value $\bar{x}$ for x_*, but with different values of the checksum. We detect MAC-collisions among those messages, and we assume that the collisions happens when processing the checksum of the internal hash function. For each such collision, we have $g(\bar{x}, K \oplus \texttt{ipad} \oplus \texttt{Sum}^{\oplus}(M)) = g(\bar{x}, K \oplus \texttt{ipad} \oplus \texttt{Sum}^{\oplus}(M'))$, and we compute the input difference $\Delta M = \texttt{Sum}^{\oplus}(M) \oplus \texttt{Sum}^{\oplus}(M')$.

Finally, we compute $g(\bar{x}, m)$ offline for a large set of random values m, and we collect collisions. Again, we compute the input difference $\Delta m = m \oplus m'$ for each collision, and we match Δm to the previously stored ΔM. When a match is found between the differences we look for the corresponding values and we have $K \oplus \texttt{ipad} \oplus \texttt{Sum}^{\oplus}(M) = m$ (or m') with high probability. This gives the value of the key K.

State recovery with a checksum. First, we note that the checksum σ does not prevent the state recovery attacks of Sections 5 and 6; the complexity only depend on the size l of the state x. Indeed, the attack of Section 5 is based on detecting collisions between pairs of messages $M_1 = m \parallel [0]^{2^{l/2}} \parallel [k] \parallel [0]^{2^{l/2}+L}$ and $M_2 = m \parallel [0]^{2^{l/2}+L} \parallel [k] \parallel [0]^{2^{l/2}}$. Since the messages have the same checksum, a collision in the state will be preserved. More generally, the attacks can easily be adapted to use only message with a fixed sum. For instance, we can use random messages with two identical blocks in the second step of the attack of Section 6, and message of the form $[i] \parallel [i] \parallel [0]^{2^s}$ in the third step: all those message have a checksum of zero.

Recovering the state of a short message. Unfortunately, the state we recover will correspond to a rather long message (*e.g.* $2^{l/2}$ or $2^{l/4}$ blocks), and all the queries based on this message will be expensive. In order to overcome this issue, we use the known state x_M after a long message M to recover the state after a short one. More precisely, we generate a set of $2^{l/4}$ long messages by appending a random message blocks m twice to M. Note that $\texttt{Sum}^{\oplus}(M \parallel m \parallel m) = \texttt{Sum}^{\oplus}(M)$. Meanwhile, we generate a set of $2^{3l/4}$ two-block messages $m_1 \parallel m_2$, with $m_1 \oplus m_2 = \texttt{Sum}^{\oplus}(M)$. We query these two sets to the HMAC oracle and collect collisions between a long and a short message. We expect that one collision correspond to a collision in the value x_* before the finalization function g. We can compute the value x_* for the long message from the known state x_M after processing M. This will correspond to the state after processing the message $m_1 \parallel m_2$ and its padding block, or equivalently, after processing the message $m_1 \parallel m_2 \parallel [2]$ (because the length block is processed with the same compression function). We can verify that the state is correctly recovered by generating a collision $m \parallel m, m' \parallel m'$ offline from the state x_*, and comparing $\texttt{HMAC}(m_1 \parallel m_2 \parallel [2] \parallel m \parallel m)$ and $\texttt{HMAC}(m_1 \parallel m_2 \parallel [2] \parallel m' \parallel m')$.

7.2 Detailed attack process

For simplicity of the description, we omit the padding block in the following description, and we refer to the previous paragraphs for the details of how to deal with the padding.

1. Recover an internal state value x_r after processing a message M_r through HMAC. Refer to Section 5 or 6 for the detailed process.
2. (online) Choose $2^{l/4}$ one-block random messages m, query $M_r \parallel m \parallel m$ to HMAC and store m and the corresponding tag.
3. (online) Choose $2^{3l/4}$ one-block random messages m, query $(\texttt{Sum}^{\oplus}(M_r) \oplus m) \parallel m$ to HMAC and look for a match between the tag value and one of the stored tag values in Step 2.
   For a colliding pair $(\texttt{Sum}^{\oplus}(M_r) \oplus m) \| m$ and $M_r \| m' \| m'$, denote $\texttt{Sum}^{\oplus}(M_r) \oplus m \| m$ as M_1 and $h(h(x_r, m'), m')$ as x_1. Generate a collision $h(h(x_1, u), u) = h(h(x_1, u'), u')$. Query $M_1 \parallel u \parallel u$ and $M_1 \parallel u' \parallel u'$ and compare the tags. If they are equal, the internal state after processing M_1 (before the checksum block) is x_1.
4. (offline) Generate $2^{3l/4}$ messages that all collide on the internal state before the checksum block by Joux's multicollision. More precisely, choose $2^{l/2}$ random message m and compute $h(x_1, m)$ to find a collision $h(x_1, m_1) = h(x_1, m'_1) = x_2$. Then iterate the same procedure to find a collision $h(x_i, m_i) = h(x_i, m'_i) = x_{i+1}$ for $i \leq 3l/4$. Denote the value of $x_{3l/4+1}$ by $\bar{x}$.
5. (online) Query the set of messages in Step 4 to HMAC in order to collect tag collisions. For each collision M and M', compute the checksum difference $\Delta M = \texttt{Sum}^{\oplus}(M) \oplus \texttt{Sum}^{\oplus}(M')$, and store $(\texttt{Sum}^{\oplus}(M), \Delta M)$.
6. (offline) Choose a set of $2^{3l/4}$ one-block random message m, compute $g(\bar{x}, m)$ and collect collisions. For each collision m and m', compute the difference $\Delta m = m \oplus m'$ and match Δm to the stored ΔM at Step 5. If a match is found, mark $\texttt{Sum}^{\oplus}(M) \oplus \texttt{ipad} \oplus m$ and $\texttt{Sum}^{\oplus}(M) \oplus \texttt{ipad} \oplus m \oplus \Delta m$ as potential key candidates.
7. (offline) filter the correct key from the potential candidates by verifying a valid message/tag pair.

7.3 Complexity and success probability analysis

We need to evaluate the complexity of our key recovery attack.

Step 1:	$O(l \cdot \log(l) \cdot 2^{l/2})$	**Step 2:**	$2^{3l/4}$	**Step 3:**	$2 \cdot 2^{3l/4}$
Step 4:	$3l/4 \cdot 2^{l/2}$	**Step 5:**	$3l/4 \cdot 2^{3l/4}$	**Step 6:**	$2^{3l/4}$
Step 7:	$O(1)$				

Overall, the fifth step dominates the complexity, and the total complexity is about $3l/4 \cdot 2^{3l/4}$ compression function computations.

Next we evaluate the success probability of our method. The first step succeeds with a probability almost 1 after several trials. Steps 2 and 3 need to guarantee a collision between a long and a short message. Since there are 2^l pairs, one such collision occurs with a probability of $1-(1-2^{-l})^{2^l} \approx 1 - 1/e \approx 0.63$. The success probability of producing no less than $2^{l/2}$ collisions at each of steps 5 and 6 is 0.5 since the expected number of collisions is $2^{l/2}$. Thus the overall success probability is no less than 0.16.

Conclusion

Our results show that the security of HMAC and hash-based MAC above the birthday bound is significantly weaker than previously expected. First, we show that distinguishing-H and state-recovery attacks are not much harder than a distinguishing-R attack, contrary to previous beliefs. Second, we show that the use of a checksum can allow a key-recovery attack against HMAC with complexity only $\tilde{O}(2^{3l/4})$. In particular, this attack is applicable to HMAC-GOST, a standardized construction.

We give a comparison of our attacks and previous attack against concrete instances of HMAC in Table 1, showing that some attacks against concrete instances are in fact less efficient than our generic attacks.

As future works, it would be interesting to find other applications of the internal state recovery for HMAC. Moreover, we expect further applications of the analysis of the functional graph, as it might be possible to use other distinguishing properties, such as the tail length, the distance of a node from the cycle, etc.

Acknowledgments

The authors would like to thank the anonymous referees for their helpful comments. Gaëtan Leurent is supported by the ERC project CRASH. Thomas Peyrin and Lei Wang are supported by the Singapore National Research Foundation Fellowship 2012 (NRF-NRFF2012-06).

References

1. Bellare, M.: New Proofs for NMAC and HMAC: Security without Collision-Resistance. In Dwork, C., ed.: CRYPTO. Volume 4117 of Lecture Notes in Computer Science., Springer (2006) 602–619
2. Bellare, M., Canetti, R., Krawczyk, H.: Keying Hash Functions for Message Authentication. In Koblitz, N., ed.: CRYPTO. Volume 1109 of Lecture Notes in Computer Science., Springer (1996) 1–15
3. Brassard, G., ed.: Advances in Cryptology - CRYPTO '89, 9th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 1989, Proceedings. In Brassard, G., ed.: CRYPTO. Volume 435 of Lecture Notes in Computer Science., Springer (1990)
4. Damgård, I.: A Design Principle for Hash Functions. [3] 416–427
5. Dolmatov, V.: GOST R 34.11-94: Hash Function Algorithm. RFC 5831 (Informational) (March 2010)
6. FAPSI, VNIIstandart: GOST 34.11-94, Information Technology Cryptographic Data Security Hashing Function (in Russian) (1994)
7. Flajolet, P., Odlyzko, A.M.: Random Mapping Statistics. In Quisquater, J.J., Vandewalle, J., eds.: EUROCRYPT. Volume 434 of Lecture Notes in Computer Science., Springer (1989) 329–354
8. Flajolet, P., Sedgewick, R.: Analytic Combinatorics. Cambridge University Press (2009)
9. Fouque, P.A., Leurent, G., Nguyen, P.Q.: Full Key-Recovery Attacks on HMAC/NMAC-MD4 and NMAC-MD5. In Menezes, A., ed.: CRYPTO. Volume 4622 of Lecture Notes in Computer Science., Springer (2007) 13–30
10. Gauravaram, P., Kelsey, J.: Linear-XOR and Additive Checksums Don't Protect Damgård-Merkle Hashes from Generic Attacks. In Malkin, T., ed.: CT-RSA. Volume 4964 of Lecture Notes in Computer Science., Springer (2008) 36–51
11. Guo, J., Sasaki, Y., Wang, L., Wu, S.: Cryptanalysis of HMAC/NMAC-Whirlpool. In: ASIACRYPT. (2013)
12. Joux, A.: Multicollisions in Iterated Hash Functions. Application to Cascaded Constructions. In Franklin, M.K., ed.: CRYPTO. Volume 3152 of Lecture Notes in Computer Science., Springer (2004) 306–316
13. Kim, J., Biryukov, A., Preneel, B., Hong, S.: On the Security of HMAC and NMAC Based on HAVAL, MD4, MD5, SHA-0 and SHA-1 (Extended Abstract). In Prisco, R.D., Yung, M., eds.: SCN. Volume 4116 of Lecture Notes in Computer Science., Springer (2006) 242–256

Table 1. Comparison of our generic attacks on HMAC, and some previous attacks on concrete hash function. We measure the complexity as the number of calls to the compression function (*i.e.* number of messages times message length)

Function	Attack	Complexity	M. len	Notes	Ref
HMAC-MD5	dist-H, state rec.	2^{97}	2		[28]
HMAC-SHA-0	dist-H	2^{100}	2		[13]
HMAC-HAVAL (3-pass)	dist-H	2^{228}	2		[13]
HMAC-SHA-1 (43 first steps)	dist-H	2^{154}	2		[13]
HMAC-SHA-1 (58 first steps)	dist-H	2^{158}	2		[20]
HMAC-SHA-1 (61 mid. steps)	dist-H	2^{100}	2		[20]
HMAC-SHA-1 (62 mid. steps)	dist-H	2^{157}	2		[20]
Generic:					
hash-based MAC (*e.g.* HMAC)	dist-H	$O(2^{l/2})$	$2^{l/2}$		Sec. 4
	state rec.	$\tilde{O}(2^{l/2})$	$2^{l/2}$		Sec. 5
	dist-H, state rec.	$O(2^{l-s})$	2^s	$s \le l/4$	Sec. 6
HMAC with a checksum	key rec.	$O(2^{3l/4})$	$2^{l/4}$		Sec. 6, 7
HMAC-MD5*	dist-H, state rec.	2^{67}, 2^{78}	2^{64}		Sec. 4, 5
		2^{96}	2^{32}		Sec. 6
HMAC-HAVAL† (any)	dist-H, state rec.	$O(2^{202})$	2^{54}		Sec. 6
HMAC-SHA-1†	dist-H, state rec.	$O(2^{120})$	2^{40}		Sec. 6
HMAC-GOST*	key rec.	2^{200}	2^{64}		Sec. 6, 7

* The MD5 and GOST specifications allow arbitrary-length messages

† The SHA-1 and HAVAL specifications limits the message length to 2^{64} bits (and 2^{64} bits is 2^{54} blocks)

14. Merkle, R.C.: One Way Hash Functions and DES. [3] 428–446
15. Naito, Y., Sasaki, Y., Wang, L., Ysuda, K.: Generic State-Recovery and Forgery Attacks on ChopMD-MAC and on NMAC/HMAC. In: IWSEC. (2013)
16. Peyrin, T., Sasaki, Y., Wang, L.: Generic Related-Key Attacks for HMAC. In Wang, X., Sako, K., eds.: ASIACRYPT. Volume 7658 of Lecture Notes in Computer Science., Springer (2012) 580–597
17. Popov, V., Kurepkin, I., Leontiev, S.: Additional Cryptographic Algorithms for Use with GOST 28147-89, GOST R 34.10-94, GOST R 34.10-2001, and GOST R 34.11-94 Algorithms. RFC 4357 (Informational) (January 2006)
18. Preneel, B.: HMAC. In van Tilborg, H.C.A., Jajodia, S., eds.: Encyclopedia of Cryptography and Security (2nd Ed.). Springer (2011) 559–560
19. Preneel, B., van Oorschot, P.C.: MDx-MAC and Building Fast MACs from Hash Functions. In Coppersmith, D., ed.: CRYPTO. Volume 963 of Lecture Notes in Computer Science., Springer (1995) 1–14
20. Rechberger, C., Rijmen, V.: New Results on NMAC/HMAC when Instantiated with Popular Hash Functions. J. UCS **14**(3) (2008) 347–376
21. Rivest, R.L.: The MD4 Message Digest Algorithm. In Menezes, A., Vanstone, S.A., eds.: CRYPTO. Volume 537 of Lecture Notes in Computer Science., Springer (1990) 303–311
22. Rivest, R.L.: The MD5 message-digest algorithm. Request for Comments (RFC) 1320, Internet Activities Board, Internet Privacy Task Force (April 1992)
23. Sasaki, Y., Wang, L.: Improved Single-Key Distinguisher on HMAC-MD5 and Key Recovery Attacks on Sandwich-MAC-MD5. In: Selected Area in Cryptography. (2013)
24. Tsudik, G.: Message Authentication with One-Way Hash Functions. In: INFOCOM. (1992) 2055–2059
25. 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`.
26. van Oorschot, P.C., Wiener, M.J.: Parallel Collision Search with Cryptanalytic Applications. J. Cryptology **12**(1) (1999) 1–28
27. Wang, L., Ohta, K., Kunihiro, N.: New Key-Recovery Attacks on HMAC/NMAC-MD4 and NMAC-MD5. In Smart, N.P., ed.: EUROCRYPT. Volume 4965 of Lecture Notes in Computer Science., Springer (2008) 237–253
28. Wang, X., Yu, H., Wang, W., Zhang, H., Zhan, T.: Cryptanalysis on HMAC/NMAC-MD5 and MD5-MAC. In Joux, A., ed.: EUROCRYPT. Volume 5479 of Lecture Notes in Computer Science., Springer (2009) 121–133
29. 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 Lecture Notes in Computer Science., Springer (2007) 355–369
30. Yu, H., Wang, X.: Full Key-Recovery Attack on the HMAC/NMAC Based on 3 and 4-Pass HAVAL. In Bao, F., Li, H., Wang, G., eds.: ISPEC. Volume 5451 of Lecture Notes in Computer Science., Springer (2009) 285–297

A Extension of the key recovery attack to more general settings

Firstly we discuss about the wide-pipe hash function case, namely $l > n$. At step 3, there become 2^{l-n} collisions on HMAC output, and thus picking out the needed collision takes $2^{l-n/2}$. At steps 5 and 6, we get more collisions

$2^{3l/2-n}$. Then at step 7, there are around 2^{2l-2n} key candidates. The total complexity is $2^{l-n/2} + 3l/4 \cdot 2^{3l/4} + 2^{2l-2n}$. Thus if and only if $n > l/2$ holds, our attack is faster than the brute-force attack. Particularly, when $n > 5l/8$ holds, the complexity is dominated by step 5, and thus is kept as $3l/4 \cdot 2^{3l/4}$.

Secondly we discuss about a generalization of the finalization of hash function; $g(x, |M|, \mathtt{Sum}^{\oplus}(M))$. It prevents us from recovering an internal state of a short message from that of a long message, since the parameter $|M|$ ia always different. Thus step 3 cannot be applied any more. We have to use rather long messages. At step 1, we restrict the length of the message to $2^{l/6}$, and apply the attack in Section 6 to recover its internal state. The complexity is $2^{5l/6}$. Accordingly, we adjust that step 4 generates $2^{2l/3}$-multicollisions, and then the complexity of step 5 is $2^{5l/6}$ since each message is $2^{l/6}$ blocks long. Finally step 6 computes $2^{5l/6}$ one random block messages. Thus the total complexity is $2^{l-n/2} + 2^{5l/6} + 2^{2l-2n}$. Particularly, when $n > 7l/8$ holds, the complexity is kept as $2^{5l/6}$.