Generic Related-key Attacks for HMAC

Thomas Peyrin[1,⋆], Yu Sasaki[2], and Lei Wang[1,3]

[1] Division of Mathematical Sciences, School of Physical and Mathematical Sciences,
Nanyang Technological University, Singapore
thomas.peyrin@gmail.com wang.lei@ntu.edu.sg
[2] NTT Secure Platform Laboratories, NTT Corporation
sasaki.yu@lab.ntt.co.jp
[3] The University of Electro-Communications

Abstract. In this article we describe new generic distinguishing and forgery attacks in the related-key scenario (using only a single related-key) for the HMAC construction. When HMAC uses a k-bit key, outputs an n-bit MAC, and is instantiated with an l-bit inner iterative hash function processing m-bit message blocks where $m = k$, our distinguishing-R attack requires about $2^{n/2}$ queries which improves over the currently best known generic attack complexity $2^{l/2}$ as soon as $l > n$. This means that contrary to the general belief, using wide-pipe hash functions as internal primitive will not increase the overall security of HMAC in the related-key model when the key size is equal to the message block size. We also present generic related-key distinguishing-H, internal state recovery and forgery attacks. Our method is new and elegant, and uses a simple cycle-size detection criterion. The issue in the HMAC construction (not present in the NMAC construction) comes from the non-independence of the two inner hash layers and we provide a simple patch in order to avoid this generic attack. Our work finally shows that the choice of the opad and ipad constants value in HMAC is important.

Key words: HMAC, hash function, distinguisher, forgery, related-key.

1 Introduction

Hash functions are among the most important basic primitives in cryptography. Informally, a hash function H is a function that takes an arbitrarily long message M as input and outputs a fixed-length hash value of size n bits. Classical security requirements are collision resistance and (second)-preimage resistance. Namely, it should be impossible for an adversary to find a collision (two distinct messages that lead to the same hash value) in less than $2^{n/2}$ hash computations, or a (second)-preimage (a message hashing to a given challenge) in less than 2^n hash computations.

Hash functions are used in many applications such as digital signatures, message integrity check and message authentication codes (MAC). 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. A MAC algorithm should also 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 not (universal forgery).

MACs are crucial for many security systems and are often implemented with the HMAC [3] algorithm, in particular for banking protocols or protocols securing Internet connections (TLS and IPSEC). HMAC was designed by Bellare *et al.* in 1996 and is now widely standardized. It has the property to use an iterative hash function as internal component (thus composed of an iterative application of a compression function) and a proof of security is given in [2]: HMAC is a pseudo-random function under the assumption that the compression function is itself a pseudo-random function.

A trivial generic extension attack exists for HMAC: by asking for enough queries to obtain an internal collision, the attacker can then add extra message blocks to generate other colliding HMAC outputs, therefore breaking the existential forgery security criterion. In order to avoid this issue, many other MACs constructions have been proposed and analyzed [28, 27, 14], reaching a security

[⋆] Supported by the Singapore National Research Foundation Fellowship 2012 (NRF-NRFF2012-06).

beyond the $n/2$ birthday bound by using bigger hash function internal state sizes. For example, the extension attack applied to an n-bit hash function with a $2n$-bit internal state requires 2^n compression function calls.

In parallel to the recent impressive advances on standardized hash function cryptanalysis, the community studied the possible impact on the security of HMAC when instantiated with these standards (such as MD5 [21] or SHA-1 [23]). There have been also some related-key analysis of HMAC instantiated with real hash functions, but no generic attack is known in this model, i.e. without using any weakness from the internal hash function used. Note that the HMAC proof [2] only holds when considering a single-key scenario and says nothing in the related-key model.

The cryptanalysts also looked at other attacks such as distinguishing-R and distinguishing-H [17]. The aim of the former is to distinguish between a random function and the HMAC construction, while the latter aims at distinguishing if the compression function used inside a HMAC construction is a random function or a specific compression function instance. It is widely believed that for the ideal narrow-pipe hash function, the distinguishing-R should require about $2^{n/2}$ computations, while distinguishing-H should require about 2^n.

Our contributions. In this article we introduce a new type of related-key distinguisher and forgery attacks for HMAC based on cycle length detection, requiring a birthday query complexity and only a single related-key. [4] The attack complexities are summarized in Table 1 together with previous work that analyzed the HMAC instantiating a dedicated hash algorithm.

Our attacks work when the inner hash function is iterative (which is the case for almost all known hash functions, and is necessary for HMAC anyway) and when a special condition is met on the key input. This condition depends on the value of the HMAC constants opad and ipad (which shows for the first time the importance in the choice of their values) and it is always fulfilled when the key length k is equal to the message input length m of the compression function. HMAC is defined to even handle cases where $k > m$ and $k = m$ is likely to happen for example with lightweight hash functions for which the total internal state size has to remain rather small. One can cite DM-PRESENT or H-PRESENT [7] hash functions (PRESENT being already an ISO standard [6]), which have respectively 80 bits and 64 bits of message input for their compression function. Also, a block cipher-based hash function using a common mode such as Davies-Meyer or Matyas-Meyer-Oseas [1] instantiated with the standardized AES [10] is also likely to meet the condition $k = m$.

We emphasize that this work is the first that exploits related-keys to attack HMAC when modeling the compression function as an ideal primitive. They are also the first attacks applying on HMAC and not on NMAC, which helps to understand the security loss when going from the latter to the former. Finally, our attacks are still applicable even when the internal hash function has a big l-bit internal state, unlike the known generic distinguishing or forgery attacks such as the extension attack. Note that many SHA-3 candidates are wide-pipe (like the finalists [16, 5, 26]) and it is the current trend in hash functions designs. Therefore, this work shows that a wide-pipe hash function used in HMAC can be weaker than the one used in simple MAC constructions such as a secret-prefix MAC and its strengthened version LPMAC [22]. In these schemes, the key (and the message length) is simply prepended to the input message, and the hash value is the MAC value. Due to the double size of the internal state, no attack is known with a smaller complexity than 2^n computations, while our attack on HMAC is more efficient, requiring only $2^{n/2+1}$ computations.

After a description of HMAC in Section 2, we introduce the generic distinguishing-R attack (requiring about $2^{n/2+1}$ computations) in Section 3, basis for the the internal state recovery attack in Section 4, the forgery attack in Section 5 and the distinguishing-H attack in Section 6. Finally, we discuss our results and propose a simple method to patch HMAC in Section 7.

[4] The weakness of such a key pair for HMAC was independently and at almost the same time pointed out by Dodis *et al.* [13, 12] to break the indifferentiability of HMAC. Note that our attacks are under the indistinguishability framework where the key value is secret to the adversary and has a uniform distribution among the key space, while the attacks in [13, 12] are under the indifferentiability framework where the key value is chosen and thus known by the adversary.

Table 1. Summary of the attack complexities

Previous attacks on `HMAC` with dedicated hash algorithm

Attack	Key Setting	Target	Size	#Rounds	Complexity	Ref.
Dist.-H	Single key	MD4	128	Full	$2^{121.5}$	[17]
Dist.-H	Single key	MD5	128	33/64	$2^{126.1}$	[17]
Dist.-H	Single Key	MD5	128	Full	2^{97}	[25]
Dist.-H	Single key	3-pass HAVAL	256	Full	$2^{228.6}$	[17]
Dist.-H	Single key	4-pass HAVAL	256	102/128	$2^{253.9}$	[17]
Dist.-H	Single key	SHA0	160	Full	2^{109}	[17]
Dist.-H	Single key	SHA1	160	43/80	$2^{154.9}$	[17]
Dist.-H	Single key	SHA1	160	50/80	$2^{153.5}$	[20]
Dist.-H	Related Key	SHA1	160	58/80	$2^{158.74}$	[20]
Inner key rec.	Single Key	MD4	128	Full	2^{63}	[9]
Inner key rec.	Single Key	SHA0	160	Full	2^{84}	[9]
Inner key rec.	Single Key	SHA1	64	34/80	2^{32}	[20]
Inner key rec.	Single Key	3-pass HAVAL	256	Full	2^{122}	[18]
Full key rec.	Single Key	MD4	128	Full	2^{95}	[15]
Full key rec.	Single Key	MD4	128	Full	2^{77}	[24]

New generic attacks on `HMAC`

Attack	Key Setting	Target	Old Generic Complexity	New Generic Complexity	Reference
Dist.-R	Related Key	Wide-pipe	$2^{l/2}$	$2^{n/2+1}$	This paper
Dist.-H	Related Key	Narrow-pipe†	2^{n}	$2^{n/2+1}$	This paper
Dist.-H	Related Key	Narrow or Wide†	2^{n}	$2^{n/2+2}+2^{l-n+1}$	This paper
Inner state rec.	Related Key	Narrow or Wide†	2^{n}	$2^{n/2+2}+2^{l-n+1}$	This paper
Ex. forgery	Related Key	Wide-pipe†	$2^{l/2}$	$2^{n/2+2}+2^{l-n+1}$	This paper

†: For a wide-pipe hash function with l-bit internal state, our attacks improve over the old generic complexity as long as $l < 2n - 1$.

2 Description of `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. When the hash function is iterative (for example see the classical Merkle-Damgård construction [19, 11]), the message M is first padded and then divided into blocks m_i of m bits each. Then, the message blocks are successively used to update an l-bit internal state cv_i (where $l \geq n$) with a compression function h: $cv_{i+1} = h(cv_i, m_i)$, and cv_0 is initialized to a fixed public value $cv_0 = IV$. Once all the message blocks have been processed, an output function g is applied to the last internal state value cv_i so as to eventually obtain $hash = g(cv_i)$. The output function therefore transforms an l-bit value into an n-bit one.

The MAC algorithm `HMAC` **[3]** is based on the `NMAC` construction that uses two k-bit keys K_{out} and K_{in}. `NMAC` replaces the public IV of a hash function $H(IV, M)$ by a secret key K to produce a keyed hash function $H(K, M)$. `NMAC` is defined by:

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

Since in practice a hash function is used as a black-box and has a fixed IV, `HMAC` simulates the keyed hash function $H(K, M)$ of `NMAC` by prepending a secret key block K to M, and computing $H(IV, K||M)$, where $||$ denotes the concatenation. Also, `HMAC` uses a single k-bit key K which is padded with zeros such that after padding the key length is equal to a multiple of m bits. 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 $k \leq m$, and thus the length of the padded key is m bits (the notation of the keys therefore denotes the padded keys). K_{in} and K_{out} are defined by: $K_{in} = K \oplus \texttt{ipad} = K \oplus \texttt{0x3636} \cdots \texttt{36}$ and $K_{out} = K \oplus \texttt{opad} = K \oplus \texttt{0x5C5C} \cdots \texttt{5C}$, where `ipad` and `opad` have the same length than a padded key. `HMAC` is defined by:

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

Since the key padding in HMAC enforces that the first compression function call(s) handles all and only the key material, we can rewrite

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

where $H_K(X)$ represents the iterative hash function H for which the initial value is changed to $h(IV, K)$.

3 Generic related-key distinguisher for HMAC

3.1 General description

Before describing our attacks, we first emphasize that for the rest of the section we will only use small n-bit messages M, such that after padding any message fit into one compression function message input. In other words, $|M||pad| = m$ and we will always compute a single compression function call in order to handle the whole message M. This is represented in Figure 1 and we have

$$\begin{aligned}\texttt{HMAC}(K, M) &= g(h(h(IV, K \oplus \texttt{opad}), g(h(h(IV, K \oplus \texttt{ipad}), M||pad))||pad)) \\ &= f_{K_{out}}(f_{K_{in}}(M))\end{aligned}$$

where $f_K(X) = g(h(h(IV, K), X||pad))$.

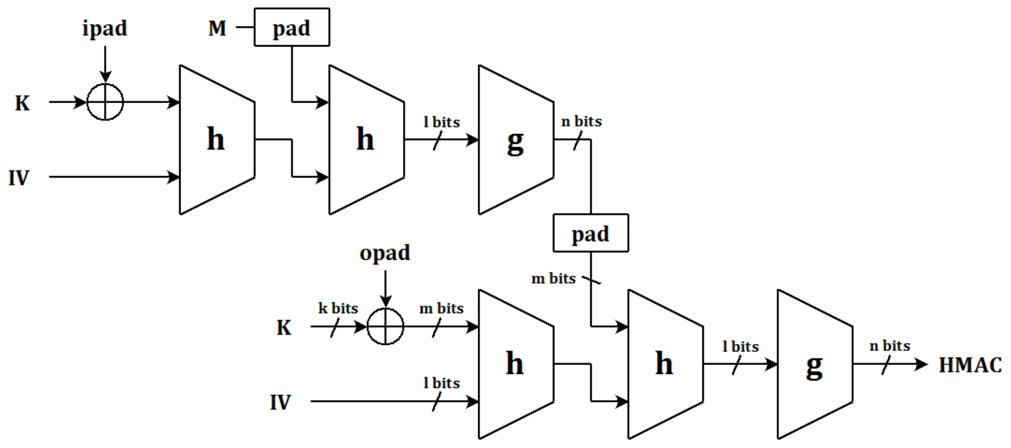


Fig. 1. The computation of HMAC with an iterated hash function when the padded message is small ($|M||pad| = m$).

The general idea underlying our attacks came from the observation that, contrary to the case of NMAC, in HMAC the inner and outer functions are not fully independent. Indeed, both inner and outer hash functions are the same function H, and the inner and outer keys are related by the relation $K_{in} \oplus K_{out} = \texttt{ipad} \oplus \texttt{opad}$.

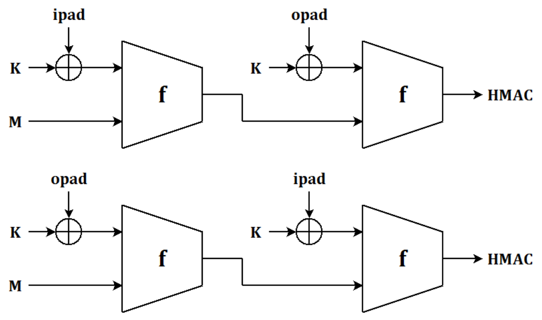
This is not an issue in the single key model, since when assuming the internal inner and outer compression functions as ideal, no information will leak on their output from this inner/outer key relation. However, in the related-key model the situation is different. When assuming that the key size k is equal to the padding size (thus one message block, i.e. $k = m$), then we can analyze what is happening when we query $\texttt{HMAC}(K, M)$ and $\texttt{HMAC}(K', M)$ with the related key $K' = K \oplus \texttt{ipad} \oplus \texttt{opad}$. For the first query the oracle will reply

$$\texttt{HMAC}(K, M) = f_{K\oplus\texttt{opad}}(f_{K\oplus\texttt{ipad}}(M)) = f_{K_{out}}(f_{K_{in}}(M))$$

and for the second query the oracle will reply

$$\begin{aligned}\mathtt{HMAC}(K', M) &= f_{K'\oplus\mathtt{opad}}(f_{K'\oplus\mathtt{ipad}}(M))\\ &= f_{K\oplus\mathtt{ipad}}(f_{K\oplus\mathtt{opad}}(M))\\ &= f_{K_{in}}(f_{K_{out}}(M))\end{aligned}$$

One can easily see that the two oracles are doing the same computation, except that `ipad` and `opad` (or K_{in} and K_{out}) are inverted. In other words, we have two oracles, one that applies $f_{K_{in}}$ and then $f_{K_{out}}$ (top figure below), and one that does the opposite $f_{K_{out}}$ and then $f_{K_{in}}$ (bottom figure below).



This non-random property seems not easy to detect since the functions $f_{K_{in}}$ and $f_{K_{out}}$ are parametrized with the secret key K, thus they are completely unknown to the attacker. However, it is possible to detect it using a cycle detection algorithm: the functions $f_{K_{in}} \circ f_{K_{out}}$ and $f_{K_{out}} \circ f_{K_{in}}$ have the same cycle structure. Indeed, it is easy to see that there is a one-to-one correspondence between each cycle from $f_{K_{in}} \circ f_{K_{out}}$ and $f_{K_{out}} \circ f_{K_{in}}$.

The attacker will start from an n-bit random input message, query the first oracle (with key K), and keep querying as new message the MAC he just received. He continues to do so for about $2^{n/2}$ queries until he gets a collision among the MACs received. This collision in fact represents a cycle in the successive computations of $f_{K_{in}} \circ f_{K_{out}}$ and this first phase defined a first walk that we denote walk A. In a second step the attacker finds also a cycle for the second oracle computations (with key $K' = K \oplus \mathtt{ipad} \oplus \mathtt{opad}$), i.e. for $f_{K_{out}} \circ f_{K_{in}}$ and that defines walk B. Finally, since the number of MACs obtained from the first and second oracle is big enough, there is a good chance that there is a collision between a MAC from walk A and an internal value of a MAC from walk B (the internal value is the output of the first hash in `HMAC`). If so, then the cycle length of the two cycles are necessarily the same since they follow exactly the same computation path starting from the collision. This is depicted in Figure 2. An attacker can use this criterion to distinguish between `HMAC` computations and a randomly chosen function, since in the latter case there is only a very low probability that the two cycles have the same length. We call the tail the part of the walk that does not belong to the cycle and we denote Z_A (resp. Z_B) the point where the tail enters the cycle for walk A (resp. walk B).

3.2 The distinguisher

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

$$Adv(\mathcal{A}) = |\Pr[\mathcal{A}(\mathtt{HMAC}_K, \mathtt{HMAC}_{K'}) = 1] - \Pr[\mathcal{A}(R_K, R_{K'}) = 1]|$$

1st phase (walk A). The attacker first chooses a random small message M_A of size n bits and initializes $q_0^A = M_A$. Then, he will query $F_K(q_0^A)$ and store the value obtained in q_1^A. He continues by querying $F_K(q_1^A)$ and by storing the answer in q_2^A, etc. for $2^{n/2} + 2^{n/2-1}$ iterations. If he observes a collision among the queries during the process, the attacker stops. If no collision is found or if the collision occurred in the $2^{n/2}$ first queries, the attacker outputs 0.

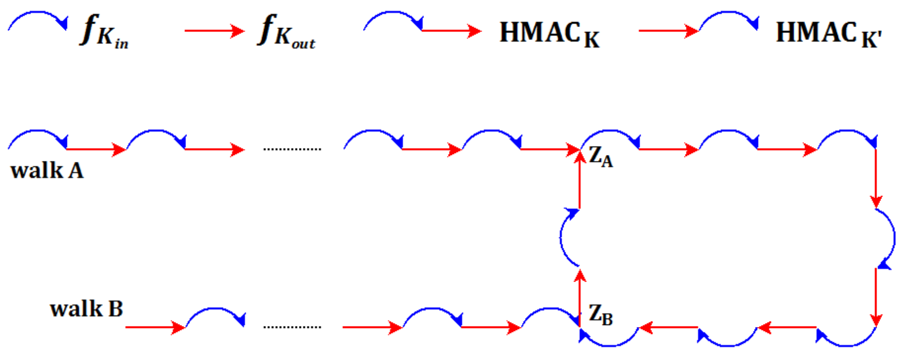


Fig. 2. The cycle structure built with access to oracles $f_{K_{out}} \circ f_{K_{in}}$ and $f_{K_{in}} \circ f_{K_{out}}$.

2nd phase (walk B). This phase is identical to the first phase, except that the attacker queries the oracle $F_{K'}$ instead of F_K. We denote q_i^B the queries asked during this phase and M_B the starting message value.

3rd phase (cycle detection). Since each query is obtained by applying the function F_K (or $F_{K'}$) on the previous query, a collision among the q_i^A (or among the q_i^B) naturally defines a cycle. If the cycle length of set A is equal to the cycle length of set B, the attacker outputs 1, otherwise he outputs 0.

3.3 Complexity and success probability

1st and 2nd phases (walk A and B). We first compute the probability that no collision is found when asking for the first $2^{n/2}$ queries in the first (or in the second) phase. In the case of randomly chosen functions:

$$P_{nc-rand} = \prod_{i=1}^{2^{n/2}} 1 - \frac{i}{2^n} \simeq \prod_{i=1}^{2^{n/2}} e^{-\frac{i}{2^n}} = e^{-2^{n/2}\cdot(2^{n/2}+1)/2^{n+1}} \simeq e^{-1/2}.$$

In the case of `HMAC` computations, a collision can occur either because of a collision on $f_{K_{in}}$ or because of a collision on $f_{K_{out}}$. Therefore, we have

$$P_{nc-hmac} = \left(\prod_{i=1}^{2^{n/2}} 1 - \frac{i}{2^n}\right)^2 \simeq \left(\prod_{i=1}^{2^{n/2}} e^{-\frac{i}{2^n}}\right)^2 = \left(e^{-2^{n/2}\cdot(2^{n/2}+1)/2^{n+1}}\right)^2 \simeq e^{-1}.$$

Then, we compute the probability that when querying the $2^{n/2-1}$ remaining elements, a collision will eventually be found in the first (or in the second) phase:

$$\begin{aligned} P_{c-rand} &= 1 - \prod_{i=1}^{2^{n/2-1}} \left(1 - \frac{2^{n/2}+i}{2^n}\right) \simeq 1 - \prod_{i=1}^{2^{n/2-1}} e^{-\frac{2^{n/2}+i}{2^n}} \\ &= 1 - e^{-2^{n/2-1}/2^{n/2} - 2^{n/2-1}\cdot(2^{n/2-1}+1)/2^{n+1}} \simeq 1 - e^{-5/8}. \end{aligned}$$

Again, in the case of `HMAC` computations, a collision can occur either because of a collision on $f_{K_{in}}$ or because of a collision on $f_{K_{out}}$. Therefore, we have

$$\begin{aligned} P_{c-hmac} &= 1 - \left(\prod_{i=1}^{2^{n/2-1}} \left(1 - \frac{2^{n/2}+i}{2^n}\right)\right)^2 \simeq 1 - \left(\prod_{i=1}^{2^{n/2-1}} e^{-\frac{2^{n/2}+i}{2^n}}\right)^2 \\ &= 1 - (e^{-2^{n/2-1}/2^{n/2} - 2^{n/2-1}\cdot(2^{n/2-1}+1)/2^{n+1}})^2 \simeq 1 - e^{-5/4}. \end{aligned}$$

To summarize, the probability of the attacker to not output 0 during both the first and second phases is equal to $(P_{nc-rand} \cdot P_{c-rand})^2 \simeq 0.079$ with randomly chosen functions and to $(P_{nc-hmac} \cdot P_{c-hmac})^2 \simeq 0.069$ with HMAC.

3rd phase (cycle detection). We need to compute the probability that the cycle found in walk A and in walk B have the same length, for both the HMAC case and the randomly chosen functions case. We denote $P_{cl-hmac}$ the former and $P_{cl-rand}$ the latter.

When the oracles are instantiated with HMAC, we already explained that $\texttt{HMAC}_K$ and $\texttt{HMAC}_{K'}$ are related by their cycle structure. If there exists a collision between a member of walk A and an internal value of a member of walk B, then we are ensured that they will enter a cycle of the same length and the attacker will output 1. Thus $P_{cl-hmac}$ is the probability that such a collision occurs. Since the first phase (resp. second phase) ensured that a collision occurs after $2^{n/2}$ queries, we are ensured that at least $2^{n/2}$ distinct elements exist in walk A (resp. walk B). Therefore, the probability $P_{cl-hmac}$ is lower bounded by

$$P_{cl-hmac} \geq 1 - \prod_{i=1}^{2^{n/2}} (1 - \frac{2^{n/2}}{2^n}) = 1 - \prod_{i=1}^{2^{n/2}} e^{-\frac{1}{2^{n/2}}} = 1 - e^{-1}.$$

Now we need to evaluate the probability $P_{cl-rand}$ that the cycles in walk A and walk B have the same length for randomly chosen functions. Since we ensured that the collision happens in the last $2^{n/2-1}$ elements instead of the first $2^{n/2}$ elements for walk A, there must exist some value z_A, $1 \leq z_A \leq 2^{n/2-1}$, such that $q^A_{2^{n/2}+z_A}$ is the first query colliding with some previous query in walk A. So the cycle length of walk A is uniformly distributed between 1 and $2^{n/2} + z_A$. Similarly for walk B, there exists a value z_B, $1 \leq z_B \leq 2^{n/2-1}$, such that the cycle length of walk B is uniformly distributed between 1 and $2^{n/2} + z_B$. Without loss of generality, let z_A be smaller than or equal to z_B. Thus, the probability that the cycles in walk A and walk B have the same length is given by

$$P_{cl-rand} = \sum_{i=1}^{2^{n/2}+z_A} \frac{1}{2^{n/2}+z_A} \times \frac{1}{2^{n/2}+z_B} < \frac{1}{2^{n/2}} \times \sum_{i=1}^{2^{n/2}+z_A} \frac{1}{2^{n/2}+z_A} = 2^{-n/2}$$

Overall the advantage of the adversary is

$$\begin{aligned} Adv(\mathcal{A}) &= |\Pr[\mathcal{A}(\texttt{HMAC}_K, \texttt{HMAC}_{K'}) = 1] - \Pr[\mathcal{A}(R_K, R_{K'}) = 1]| \\ &\geq |(P_{nc-hmac} \cdot P_{c-hmac})^2 \cdot P_{cl-hmac} - (P_{nc-rand} \cdot P_{c-rand})^2 \cdot P_{cl-rand}| \\ &\simeq (e^{-1} \cdot (1 - e^{-5/4}))^2 \cdot (1 - e^{-1}) = 0.044 \end{aligned}$$

and it can be increased towards $(1 - e^{-1}) = 0.63$ by allowing the attacker to spend a bit more computations in the first and second phases (instead of outputting 0, he just starts the phase over until he succeeds).

The complexity of the distinguisher is about $2^{n/2} + 2^{n/2-1}$ computations for each of the first and second phase, thus about $2^{n/2+1}$ computations in total.

Note that one could argue that it is possible to find a distinguisher for HMAC in the single-key model using $2^{n/2}$ queries, just by observing that HMAC will produce twice more collisions than a random function, since HMAC is composed of the iteration of two functions (this is reflected in the gap we have between $P_{nc-rand}$ and $P_{nc-hmac}$, or between P_{c-rand} and P_{c-hmac}). However, this distinguisher will be invalid because it hides the fact that one call to HMAC requires two random function calls, and then for the same cost the attacker could have called twice more the random functions R_K or $R_{K'}$, thus the advantage between $P_{nc-rand}$ and $P_{nc-hmac}$, or between P_{c-rand} and P_{c-hmac} vanishes. Alternatively, we could define a slightly different distinguishing-R game in which the attacker has to distinguish between $(\texttt{HMAC}_K, \texttt{HMAC}_{K'})$ and $(R_K, R_{K'})$, and where $R_K = R^1_K \circ R^0_K$ and $R_{K'} = R^1_{K'} \circ R^0_{K'}$ with R^0_K, R^1_K, $R^0_{K'}$, $R^1_{K'}$ being four independently chosen random functions. Of independent interest, we found this new game is quite natural and fills a notion gap between the distinguishing-R game and the distinguishing-H game. Recall that a distinguishing-R game is to distinguish HMAC from a random oracle, and a distinguishing-H game is to distinguish a known compression function from a fixed input length random function under the constraint that they are used in HMAC based on a known domain extension hash algorithm. Surprisingly, there is no notion of distinguishing a known domain extension hash algorithm from a random oracle under the constraint that they are both used in HMAC, which we propose to name distinguishing-D. Our new game exactly covers this notion in a related-key setting.

3.4 Implementation

We implemented this distinguisher on HMAC instantiated with SHA-2 truncated to 32 bits. In this scenario, we have $n = 32$, $l = 256$, $m = k = 512$ and the best distinguishing-R attack previously known (the extension attack) requires 2^{128}. Our method requires only 2^{17} computations and takes less than a second to perform on a modern PC. We verified experimentally the validity of the probabilities computed theoretically.

As a proof of concept, we provide an example of two walks A and B that have the same cycle length. The key K was chosen randomly and $K' = K \oplus \texttt{ipad} \oplus \texttt{opad}$. The message M was also chosen randomly and is the starting query for both walk A and walk B. Finally, both walk have the same cycle length of 79146 elements.

```
K  = 67ae0a7c e69eda19 35d12aa1 7eab84ed 4b161697 0cfc317d 95d0cc42 9d06fecd
     419788c8 7e7e4922 ba3dc78d 3c59ad01 afd94837 a3cb082f 0cbf05ab c14b78b2
K' = 0dc46016 8cf4b073 5fbb40cb 14c1ee87 217c7cfd 66965b17 ffbaa628 f76c94a7
     2bfde2a2 14142348 d057ade7 5633c76b c5b3225d c9a16245 66d56fc1 ab2112d8
M  = 56753af1
```

4 Internal state recovery attack

In this section we extend the distinguisher from Section 3 and we present an internal-state-recovery attack that will be useful for the latter sections showing forgery and distinguishing-H attacks. These attacks are applicable to both narrow-pipe and wide-pipe hash functions under some conditions. As an example for a narrow-pipe hash function without finalization $g(\cdot)$, i.e. SHA-256 and SHA-512 [23], these attacks achieve a birthday-bound complexity $2^{n/2}$, thus significantly reducing the expected complexity of 2^n.

4.1 General idea

We observe that if walk A and walk B follow the structure in Figure 2, then for any query in the cycle of walk A, denoted as q^A, the inner hash value $H_{K_{in}}(q^A)$ is necessarily equal to some query in the cycle of walk B, denoted as q^B. The goal is therefore to find this query among all $\#q^B$ candidate values (all the members of walk B that belong to the cycle). In other words, we would like to synchronize the two cycles from walk A and walk B, which we already know have the same length.

In general, even if we know that walk A and walk B have the same length and are actually doing the same computations, it seems hard to synchronize the two cycles because we do not know where the tail in walk A and in walk B is entering the cycle. However, in the special case where the collision between walk A and walk B happens in the tail (and not in the cycle), then we know that the tails are entering the cycle at the same position (see Figure 3). In that case, the cycles are directly synchronized and the attacker knows all the successive hash output values for every computation in the cycle (he knows the output values of all the $H_{K_{in}}$ and $H_{K_{out}}$ computed inside the cycle).

The first and second phases of the attack will be devoted to building a walk A and walk B with a rather long tail, such that during the third phase there is a good chance to get a collision between an element of the tail of walk A and an element of the tail of walk B. In order to recover an internal state, he will focus on one randomly chosen value belonging to the cycle, denoted q^A,

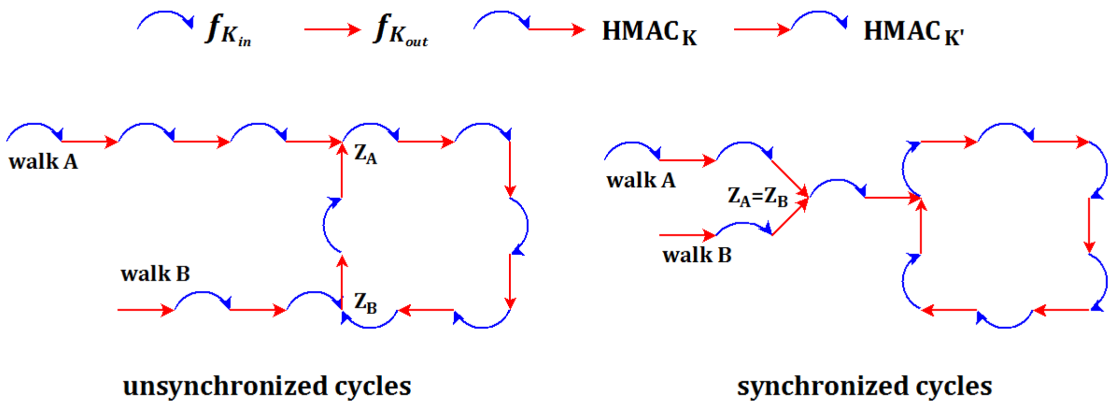


Fig. 3. Two walks A and B colliding and sharing a cycle. The left example shows unsynchronized cycles (the collision happens in the cycle, thus $Z_A \neq Z_B$), the right shows synchronized cycles (the collision happens before the cycle, in the tails, thus $Z_A = Z_B$).

and its next hash output q^B, with $q^B = H(K_{in}, q^A)$. Then he will try to guess the internal hash value $X = h(h(IV, K_{in}), q^A||pad_1)$ that led to q^B, i.e. $g(X) = q^B$.

We assume that $g(\cdot)$ is easy to invert (given an output u, it is easy to find all preimages leading to u) and that it is balanced (given an output value, there exists 2^{l-n} corresponding input values through g). Inverting g provides 2^{l-n} candidates X_i such that $g(X_i) = q^B$. For each of these candidates, we will apply a filter to remove the bad guesses. The filter is based on an offline extension of the computation of $H_{K_{in}}$

4.2 Detailed procedure

1st phase (walk A). The attacker chooses a random small message M_A of size n bits and initializes $q_0^A = M_A$. Then he will query $\texttt{HMAC}_K(q_0^A)$ and store the value obtained in q_1^A. He continues by querying $\texttt{HMAC}_K(q_i^A)$, and by storing the answer in q_{i+1}^A for $i = 0, 1, \ldots, 2^{n/2}$. If no cycle is generated (no collision among the queries q_i^A) or if the walk A generated has a tail smaller than $2^{n/2-2}$, then the attacker chooses another random n-bit message as starting query q_0^A and repeats the search procedure until a walk A with a cycle and a tail of at least $2^{n/2-2}$ elements are found.

We evaluate the success probability of finding a proper walk A by trying one set of $2^{n/2}$ iterative queries. First we would like the first $2^{n/2-1}$ elements be distinct and the probability of this event is approximately $e^{-1/8}$ (the evaluation is similar to the one from Section 3, thus we omitted it here). Then the probability that the last $2^{n/2-1}$ queries produce a cycle is approximately $e^{-3/8}$. We evaluate the probability that the tail of walk A has at least $2^{n/2} - 2$ elements. Note that we have guaranteed that the query q_i^A causing the first collision happens during the i-th iteration, with $i > 2^{n/2} - 1$. Therefore, the probability that q_i^A does not collide with the first $2^{n/2} - 2$ elements is $1 - (2^{n/2-2}/i) \geq 1/2$. Finally, we conclude that by trying one set of $2^{n/2}$ iterative queries, the success probability of generating a proper walk A is at least $e^{-1/8} \times e^{-3/8} \times 1/2 \simeq 0.303$.

2nd phase (walk B.) The procedure is identical to the first phase except that the attacker is querying $\texttt{HMAC}_{K'}$ with $K' = K \oplus \texttt{ipad} \oplus \texttt{opad}$ instead of $\texttt{HMAC}_K$. He obtains a walk B that has a cycle and whose tail contains at least $2^{n/2-2}$ elements with probability of about 0.303 (identical to 1st phase).

3rd phase (collision). The attacker checks that there is a collision between an element from walk A and one from walk B, which can be done by verifying that walk A and walk B have the same cycle length. He also wants this collision to happen more exactly between a member of the tail of walk A and a member of the tail of walk B. This event happens with probability $1 - e^{-1} \simeq 0.63$

and if such a collision occurs, then the cycles from walk A and walk B are synchronized. In other words, the attacker knows that the tail in walk A entered the cycle at the same position that the tail in walk B entered its own cycle and as a consequence he knows all the succesive internal values for the $\texttt{HMAC}_K$ and $\texttt{HMAC}_{K'}$ computations belonging to the cycle. We denote q^A, q^B and q^C three consecutive internal states, that is $q^B = H(K_{in}, q^A)$, $q^C = H(K_{out}, q^B)$ and $q^C = \texttt{HMAC}_K(q^A)$.

4th phase (recovery by filtering.) Given q^A, q^B and q^C, known by the attacker, the goal is now to recover the inner hash function internal state just before applying the output function g. In other words, the attacker is trying to recover $X = h(h(IV, K_{in}), q^A||pad_1)$, with $g(X) = q^B$. He first inverts the output function g from q^B and gets 2^{l-n} candidate values X_j.

The attacker chooses $2^{n/2}$ random distinct messages M_i, $0 \le i < 2^{n/2}$, such that we have that each $q^A||pad_1||M_i||pad_2$ fits into exactly two message blocks. He queries the messages $q^A||pad_1||M_i$ to $\texttt{HMAC}_K$ and look for collisions among the outputs. A collision happens in inner hash with a probability $1 - e^{-1/2}$. At the same time, we want to avoid faulty collision, i.e. collision in the outer hash instead of the inner hash, and this happens with probability $e^{-1/2}$. We denote (M, M') the pair of colliding message found and the success probability is $(1 - e^{-1/2}) \times e^{-1/2} \simeq 0.23$.

For each of the 2^{l-n} candidate values X_j, the attacker computes the two values $g(h(X_j, M||pad_2))$ and $g(h(X_j, M'||pad_2))$, and checks whether they are equal. If it is the case, the attacker stores X_j as a very likely candidate for the yet unknown value of X. Since there are in total 2^{l-n} candidate values, and the filter is of n-bit, 2^{l-2n} candidates will be stored. The attacker repeats the colliding messages (M, M') search and the filtering process until only one candidate, namely the real value of X, is left.

Overall, the complexity of the attack is less than $2^{n/2+2}$ queries, and 2^{l-n+1} offline computations. The success probability is around $0.303 \times 0.303 \times 0.63 \times 0.23 = 0.013$. By repeating the phases from 2 and 4 several times, the success probability will be increased.

5 Forgery attacks

This section describes the related-key forgery attacks on $\texttt{HMAC}$. The adversary is given access to two oracles $\texttt{HMAC}_K$ and $\texttt{HMAC}_{K'=(K\oplus\texttt{ipad}\oplus\texttt{opad})}$. After interacting with $\texttt{HMAC}_K$ and $\texttt{HMAC}_{K'}$, he outputs a message and MAC value (M, σ), such that the message has not be queried for $\texttt{HMAC}_K$. If σ is a valid MAC value for M through $\texttt{HMAC}$ with key K, the adversary is said to have successfully forged M for $\texttt{HMAC}_K$. More precisely, when the attacker is free to choose M it is an existential forgery, while if the message is fixed by the challenger beforehand it is a universal forgery.

A commonly known generic existential forgery attack on $\texttt{HMAC}$ (even in the single-key setting) is the so-called extension attack. The attacker first searches for a pair of messages (M, M') colliding on the last l-bit internal state of the inner hash (just before the application of the output function g in the inner hash function call), then appends each of them with the same additional message block X. Since the last internal state is the same for both messages (M, M'), the two computations of this extra message block X will also behave identically. Finally, by querying the $\texttt{HMAC}$ value for one of the two message $M||X$, the attacker directly forge the other one $M'||X$ by outputting the same MAC value. The complexity of this existential forgery attack is around $2^{l/2}$ queries.

We extend the internal-state-recovery attack from Section 4 to an existential forgery attack. The method is simple. Following the procedure in Section 4, the attacker first recovers the internal state X during the $\texttt{HMAC}_K$ computation of one of the n-bit messages queried and we denote this message by M. Then, using about $2^{n/2}$ computations, he generates offline a pair of distinct messages M' and M'' of the same length satisfying $g(h(X, M'||pad_2)) = g(h(X, M''||pad_2))$, where pad_2 stands for the padding appended to the message $M||M'$ (or $M||M''$) when applying the hash function H. Finally, the attacker queries $M||pad_1||M'$ to the oracle $\texttt{HMAC}_K$ and receives a value T', where pad_1 stands for the padding added to the message M when applying the hash function H. He can forge the MAC value T'' for the message $M||pad_1||M''$ through $\texttt{HMAC}_K$ since $T'' = T'$. The overall complexity of this attack is $2^{n/2+2}$ queries and $2^{l-n} + 2^{n/2}$ computations. Note that in particular for the case $l < 2n$, our attack is faster than the commonly known existential forgery attack requiring $2^{l/2}$ computations.

One can trivially extend this existential forgery attack to an "almost-universal" forgery attack, where the attacker can only choose the first block and the $l/2$ first bits of the second block of the message to be forged. In practice, this would be very close to a universal forgery if one assumes that a few bytes of data in the header of the messages to be MACed can be controlled by the attacker.

6 Distinguishing-H attacks

This section proposes two distinguishing-H attacks in the related-key setting. The attacker is given access to two oracles $\mathtt{HMAC}_K$ and $\mathtt{HMAC}_{K'}$ with $K' = K \oplus \mathtt{ipad} \oplus \mathtt{opad}$. The compression function of the $\mathtt{HMAC}$ oracles is instantiated either with a known dedicated compression function h or with a random chosen function r from $\mathcal{F}_l^{m+l}$ (the set of $(m+l)$-bit to l-bit functions), which we denote $(\mathtt{HMAC}^h_K, \mathtt{HMAC}^h_{K'})$ and $(\mathtt{HMAC}^r_K, \mathtt{HMAC}^r_{K'})$ respectively. The goal of the adversary is to distinguish between the two cases and its advantage is given by

$$Adv(\mathcal{A}) = \left|\Pr[\mathcal{A}(\mathtt{HMAC}^h_K, \mathtt{HMAC}^h_{K'}) = 1] - \Pr[\mathcal{A}(\mathtt{HMAC}^r_K, \mathtt{HMAC}^r_{K'}) = 1]\right| .$$

6.1 Distinguishing-H attack I: comparing cycles lengths

The distinguisher in Section 3 can be extended to a distinguishing-H attack, as long as the finalization $g(\cdot)$ is bijective and invertible, for example the identity function. Without loss of generality, we omit the output function g. The only difference from the distinguisher in Section 3 will be that in order to produce walk A and walk B we will make full-block long iterative queries, namely m-bit queries, instead of n-bit queries. A graphical view of one iteration in a walk is given in Figure 4. Let pad_1 be the padding to an n-bit message and pad_2 the padding to an m-bit message. The attacker first chooses a small random n-bit value q_0^A. He then queries $q_0^A||pad_1$ to $\mathtt{HMAC}_K$ and receives X_0. He computes $h(X_0, pad_2)$ offline and stores the output as q_1^A. He continues to query $q_i^A||pad_1$, receive X_i and apply $h(X_i, pad_2)$ offline to produce q_{i+1}^A. With the same process, the attacker produces walk B, except that he queries $\mathtt{HMAC}_{K'}$ instead of $\mathtt{HMAC}_K$.

If $\mathtt{HMAC}$ oracles are instantiated with h, then $h(\mathtt{HMAC}_K(\cdot), pad_2)$ is $f_{K_{in}} \circ f_{K_{out}}$ and $h(\mathtt{HMAC}'_K(\cdot), pad_2)$ is $f_{K_{out}} \circ f_{K_{in}}$, where $f_{K_{in}}$ and $f_{K_{out}}$ are defined in Figure 4. So walk A and walk B have a good chance to have the structure explained in Section 3 and depicted in Figure 2, leading to cycles of equal length. On the other hand, if $\mathtt{HMAC}$ oracles are instantiated with r, walk A and walk B are independent. Thus by detecting the cycles lengths, the adversary can distinguish $(\mathtt{HMAC}^h_K, \mathtt{HMAC}^h_{K'})$ from $(\mathtt{HMAC}^r_K, \mathtt{HMAC}^r_{K'})$. The complexity and the success probability are similar to the ones for the distinguisher in Section 3.

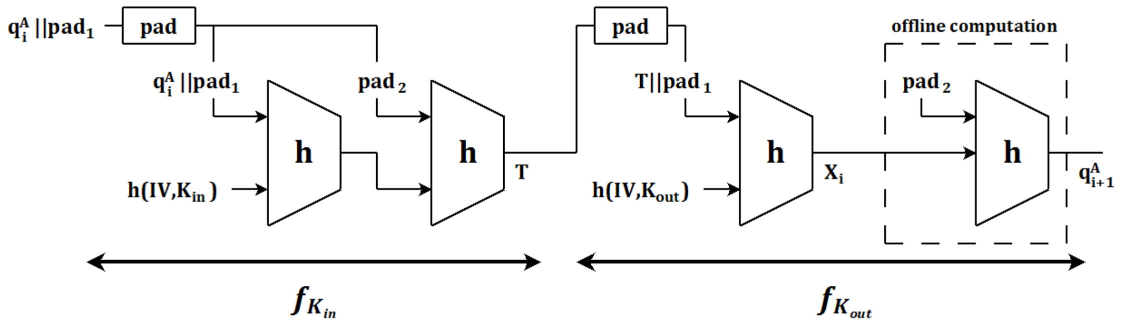


Fig. 4. Distinguisher-H attack I.

6.2 Distinguishing-H attack II: recovering internal state

The internal state recovery attack in Section 4 can be extended to a distinguishing-H attack as well. The adversary first regards the HMAC oracles as $(\mathtt{HMAC}^h_K, \mathtt{HMAC}^h_{K'})$, and applies the internal state recovery procedure from Section 4 to obtain an internal state value X of some n-bit query q^A in a walk. Then he searches offline a pair of distinct messages (M, M') satisfying $g(h(X, M)) = g(h(X, M'))$, which costs $2^{n/2}$ computations. Finally, he queries $\mathtt{HMAC}_K$ with $q^A||pad_1||M$ and $q^A||pad_2||M'$ to check whether the two MAC values collide. If they do the attacker outputs 1, otherwise he outputs 0.

If the compression function is h, the probability that $\mathtt{HMAC}_K(q^A||pad_1||M)$ collides with the value $\mathtt{HMAC}_K(q^A||pad_1||M')$ is equal to the success probability of recovering X in the attack of Section 4. If the hash function is r, the probability that $\mathtt{HMAC}_K(q^A||pad_1||M) = \mathtt{HMAC}_K(q^A||pad_1||M')$ is negligible.

Overall, the complexity is $2^{n/2+2}$ queries, $2^{l-n+1} + 2^{n/2}$ offline computations and the success probability is 0.013.

7 Patching HMAC and discussions

We emphasize again that the related-key issue depicted in this article only exists when the attacker can query $f_{K_{out}} \circ f_{K_{in}}$ and $f_{K_{in}} \circ f_{K_{out}}$ with related-key relations, and therefore keep the two computation chains synchronized if a collision happens. In the case of HMAC this is possible only when $k = m$ or $k = m - 1$ since the last bit of ipad and opad are equal (otherwise, for a smaller key the attacker can not build a proper related-key). This shows that **the choice of ipad and opad is not anecdotal**. For example, if ipad and opad were very similar, then our attacks would work for basically any key length. Also, we observe that our attacks are the first to apply to HMAC and not to NMAC, thus helping the community to understand what security we loose when going from NMAC to HMAC.

Even if our attack is only theoretical due to its high birthday complexity, it is interesting to study how one can patch the scheme and avoid this related-key issue. Since one of the best feature from HMAC is that it uses a hash function as a black box, without any need to change the primitive implementation, our goal is to find a patch that does not affect the hash function definition. Indeed, an easy and efficient tweak would be for example to force different IVs for the inner and outer instances of H in HMAC, but that would require modifying H's implementation. We note that truncating the output of HMAC would also work (the attacker would have to successively guess the truncated bits for each received query in order to continue the computation chain), but we do not consider this solution as satisfactory because reducing the output length will directly reduce the expected generic security of the MAC algorithm.

A first try could be to xor some distinct constants to the inner and/or outer hash message input in an attempt to separate the $f_{K_{out}}$ and $f_{K_{in}}$ computations. However, with such a patch, an attacker can adapt his query strategy and still perform a modified version of the attack from Section 3 to maintain the computation chains synchronized.

Our proposed solution is instead to force an extra fixed bit (or byte) before the input message M. This patch would not harm much the efficiency of the scheme since only one bit (or one byte) would be added to the message to hash for the inner hash function call (actually the efficiency will be the same if the message plus one bit still fit in the same number of message blocks). Also, this patch can even be applied on top of HMAC, as a preprocessing phase before calling the primitive, thus allowing to use existing HMAC libraries without having to modify them.

The related-key distinguishing-R attack from Section 3 is thwarted because now the inner and outer function are made distinct, even when querying with keys K and $K' = K \oplus \mathtt{opad} \oplus \mathtt{ipad}$. The attacker can no more adapt the queries to circumvent this countermeasure and keep the computation chains synchronized. The security proofs of HMAC still hold with this patch since it is trivial to see that any attack on this new proposal will also apply on HMAC.

Note that adding this extra bit (or byte) to the input of the outer hash function instead of the inner one, in an attempt to not reduce the efficiency (in most cases the hash function output size n is much smaller than its message input size m and fit in one block, thus the efficiency would actually be very likely to remain exactly the same), would not prevent the attack from Section 3 to be applicable, since the attacker could simply adapt his query strategy: instead of getting a value V from the HMAC oracle and then query this value V again etc., he could simply prepend a 0 to the received query $0||V$ before querying it again and eventually get the K and K' computations synchronized again.

We observed that appending or prepending the extra bit to the message have actually different impact on the security. For the former, the distinguishing-H attack (approach I) from Section 6 can still apply in the case of a narrow pipe internal hash function, while for the latter the attacker can no more play with pad_2 to absorb the prepended bit. Thus, **our final proposal is to simply prepend a 0 bit (or byte) to the input message of HMAC** . Namely, this new version HMAC' would be defined as

$$\mathtt{HMAC}'(K,M) = H_{K\oplus\mathtt{opad}}(H_{K\oplus\mathtt{ipad}}(0||M)) = H_{K_{out}}(H_{K_{in}}(0||M)) = \mathtt{HMAC}(K, 0||M)$$

Taking in account the fact that the related-key attacks described in this article only work for special key length, we propose to apply our patch to HMAC only when $k = m$ or $k = m - 1$.

We leave as an open problem to find a patch that has no impact on the efficiency (not even a single bit), without modifying the implementation of the hash function H (thus without using distinct IVs for the outer and inner hash calls).

As a final remark, we observe that for HMAC one should only consider related-keys of the same length than the original key. Indeed, for HMAC one can easily check that when the length of the key K is not a multiple of m, then the key $K' = K||0$ is equivalent to K in the sense that $\mathtt{HMAC}_K(M) = \mathtt{HMAC}_{K'}(M)$ for any message M (this related-key relation is even valid in the formalization of related-key attacks from Bellare and Kohno [4] since no two different keys have the same related-key). This is due to the fact that the padding of the key (so that its length becomes a multiple of m) is weak and do not distinguish between keys of different length. A possible patch in order to avoid any equivalent key would to simply pad the key with a 1 and as many zeros as needed (possibly none) such that $K||10\ldots0$ is a multiple of m, instead of the original $0\ldots0$ padding.

Conclusion

In this article we introduced a new type of distinguishing-R, distinguishing-H, internal state recovery and forgery attacks for HMAC in the related-key setting. While the applicability of this attack is only theoretical, it uses a novel attack angle, the cycle length. It is the first attack that applies on HMAC and not on NMAC and it provides a better understanding of the role of the constants ipad and opad. We also showed that our attacks can be avoided with a simple patch that only prepends 1 bit or 1 byte to the head of a message.

References

1. A. Menezes, P. van Oorschot, and S. Vanstone. CRC-Handbook of Applied Cryptography. CRC Press, 1996.
2. 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.
3. 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.
4. Mihir Bellare and Tadayoshi Kohno. A Theoretical Treatment of Related-Key Attacks: RKA-PRPs, RKA-PRFs, and Applications. In Eli Biham, editor, *EUROCRYPT*, volume 2656 of *Lecture Notes in Computer Science*, pages 491–506. Springer, 2003.
5. Guido Bertoni, Joan Daemen, Michal Peeters, and Gilles Van Assche. Keccak specifications. Submission to NIST, 2008. `http://keccak.noekeon.org/Keccak-specifications.pdf`.
6. Andrey Bogdanov, Lars R. Knudsen, Gregor Leander, Christof Paar, Axel Poschmann, Matthew J. B. Robshaw, Yannick Seurin, and C. Vikkelsoe. PRESENT: An Ultra-Lightweight Block Cipher. In Pascal Paillier and Ingrid Verbauwhede, editors, *CHES*, volume 4727 of *Lecture Notes in Computer Science*, pages 450–466. Springer, 2007.

7. Andrey Bogdanov, Gregor Leander, Christof Paar, Axel Poschmann, Matthew J. B. Robshaw, and Yannick Seurin. Hash Functions and RFID Tags: Mind the Gap. In Elisabeth Oswald and Pankaj Rohatgi, editors, *CHES*, volume 5154 of *Lecture Notes in Computer Science*, pages 283–299. Springer, 2008.
8. Gilles Brassard, editor. *Advances in Cryptology - CRYPTO '89, 9th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 1989, Proceedings*, volume 435 of *Lecture Notes in Computer Science*. Springer, 1990.
9. 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.
10. Joan Daemen and Vincent Rijmen. *The Design of Rijndael: AES - The Advanced Encryption Standard.* Springer, 2002.
11. Ivan Damgård. A Design Principle for Hash Functions. In Brassard [8], pages 416–427.
12. Yevgeniy Dodis, Thomas Ristenpart, John Steinberger, and Stefano Tessaro. To Hash or Not to Hash Again? (In)differentiability Results for H^2 and HMAC. Cryptology ePrint Archive, Report 2013/382, 2013. `http://eprint.iacr.org/2013/382`.
13. 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.
14. Yevgeniy Dodis and John P. Steinberger. Domain Extension for MACs Beyond the Birthday Barrier. In Kenneth G. Paterson, editor, *EUROCRYPT*, volume 6632 of *Lecture Notes in Computer Science*, pages 323–342. Springer, 2011.
15. 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.
16. Praveen Gauravaram, Lars R. Knudsen, Krystian Matusiewicz, Florian Mendel, Christian Rechberger, Martin Schläffer, and Søren S. Thomsen. Grøstl- a SHA-3 candidate. Submitted to NIST, 2008. `http://www.groestl.info`.
17. 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*. Springer, 2006.
18. 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.
19. Ralph C. Merkle. One Way Hash Functions and DES. In Brassard [8], pages 428–446.
20. Christian Rechberger and Vincent Rijmen. New Results on NMAC/HMAC when Instantiated with Popular Hash Functions. *J. UCS*, 14:347–376, 2008.
21. Ronald L. Rivest. The MD5 message-digest algorithm. Request for Comments (RFC) 1320, Internet Activities Board, Internet Privacy Task Force, April 1992.
22. Gene Tsudik. Message Authentication with One-Way Hash Functions. In *ACM SIGCOMM Computer Communication Review*, volume 22(5), pages 29–38, 1992.
23. 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`.
24. 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.
25. 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.
26. Hongjun Wu. The Hash Function JH. Submitted to NIST, 2008. `http://icsd.i2r.a-star.edu.sg/staff/hongjun/jh/jh.pdf`.
27. Kan Yasuda. Multilane HMAC - Security beyond the Birthday Limit. In K. Srinathan, C. Pandu Rangan, and Moti Yung, editors, *INDOCRYPT*, volume 4859 of *Lecture Notes in Computer Science*, pages 18–32. Springer, 2007.
28. Kan Yasuda. A Double-Piped Mode of Operation for MACs, PRFs and PROs: Security beyond the Birthday Barrier. In Antoine Joux, editor, *EUROCRYPT*, volume 5479 of *Lecture Notes in Computer Science*, pages 242–259. Springer, 2009.