Cryptanalysis of HMAC/NMAC-Whirlpool

Jian Guo[1], Yu Sasaki[2], Lei Wang[1], and Shuang Wu[1]

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

Abstract. In this paper, we present universal forgery and key recovery attacks on the most popular hash-based MAC constructions, *e.g.*, HMAC and NMAC, instantiated with an AES-like hash function Whirlpool. These attacks work with Whirlpool reduced to 6 out of 10 rounds in single-key setting. To the best of our knowledge, this is the first result on "original" key recovery for HMAC (previous works only succeeded in recovering the equivalent keys). Interestingly, the number of attacked rounds is comparable with that for collision and preimage attacks on Whirlpool hash function itself. Lastly, we present a distinguishing-H attack against the full HMAC- and NMAC-Whirlpool.

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

1 Introduction

AES (Advanced Encryption Standard) [6] is the probably most used block cipher nowadays, and it also inspires many designs for other fundamental primitives of modern cryptography, *e.g.*, hash function. As cryptographic algorithms for security applications, AES and AES-like primitives should receive continuous security analysis under various protocol settings. This paper discusses the security evaluation of these primitives in one notable setting; the MAC (Message Authentication Code) setting.

A MAC is a symmetric-key construction to provide integrity and authenticity for data. There are two popular approaches to build a MAC. The first approach is based on a block cipher or a permutation, *e.g.*, the well-known CBC (Cipher Block Chaining) MAC [1]. Such designs with an AES-like block cipher (or permutation) include CMAC-AES [28], PC-MAC-AES [19], ALPHA-MAC [7] and PELICAN-MAC [8]. A series of analysis results have been published on these AES-like block ciphers (or unkeyed permutations) under the CBC MAC setting. Refer to [12, 13, 32, 4, 9]. From a high-level view, cryptanalysts have managed to extend several analysis techniques devised on block cipher itself to also work in the CBC MAC setting, *e.g.*, [32, 9] use the impossible differential attack. The second approach is based on a hash function. Such designs with an AES-like hash function include HMAC-Whirlpool and HMAC-Grøstl. Surprisingly, there is NO algorithmic analysis result yet on these AES-like hash functions in the MAC setting to our best knowledge, though a side-channel attack was published on HMAC-Whirlpool [33].

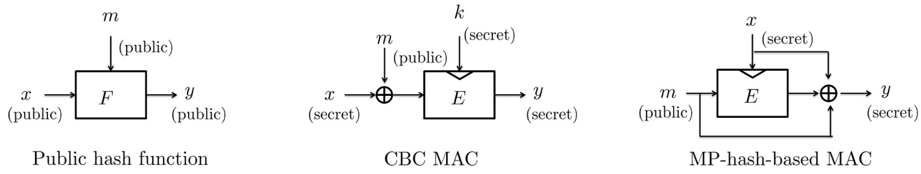

Fig. 1. Comparison of attack models

We briefly discuss the difficulty of applying the analysis techniques, which are devised to analyze public AES-like hash functions or to analyze AES-like block ciphers in the CBC MAC setting, to evaluate AES-like hash functions under the hash-based MAC setting. More precisely, we make a comparison of their model from an attacker's view by focusing on the underlying iterated small primitives; compression function of a hash function and block cipher of CBC MAC, which is also explained in Figure 1. A few new notations are introduced here: x is an internal state after processing previous message blocks, m is a current message block, y is an updated internal state, k is a secret key of block cipher, F is a compression function, and E is a block cipher.

For a hash function in public setting and in MAC setting, the main difference from an attacker's view is that x and y are public in the former setting, but are secret in the latter setting. Note that the effective analysis techniques rebound attack [18] and splice-and-cut preimage attack [25] on AES-like hash functions in public setting use a start-from-the-middle approach, which requires to know and to control the internal values of the compression function, and thus requires that x is public to the attacker. Therefore these techniques cannot be applied trivially in MAC setting.

For CBC MAC and hash-based MAC, the main difference is how a message block is injected to an internal state. CBC MAC uses a simple XOR sum $x \oplus m$, while hash-based MAC usually compresses x and m in a complicated process, *e.g.*, the Miyaguchi-Preneel (MP) scheme $E_x(m) \oplus m \oplus x$. It affects the applicability of differential cryptanalysis. The attacker is able to derive the internal state difference Δx in the CBC MAC setting (*i.e.*, randomize message block m to find a pair m and m' that leads to a collision on the input to E detectable from the colliding MAC outputs, and derive $\Delta x = m \oplus m'$). On the other hand, the internal state difference cannot be derived in the hash-based MAC setting except the collision case $\Delta x = 0$, which sets a constraint on the differentials of the underlying block cipher that can be exploited by an attacker.

This paper gives the first step on the algorithmic security evaluation of AES-like hash functions in the hash-based MAC setting. The main attack target is the Whirlpool hash function in the HMAC setting, which is motivated by the fact that both schemes are internationally standardized.

Whirlpool [24] was proposed by Barreto and Rijmen in 2000. Its compression function is built from an AES-like block cipher following Miyaguchi-Preneel mode. Whirlpool has been standardized by ISO/IEC, and has been implemented in many cryptographic software libraries such as FreeOTFE and TrueCrypt. Its security has been evaluated and approved by NESSIE [20]. The first cryptanalysis result was published by Mendel *et al.* in 2009 [18], which presented a collision attack on 4-round Whirlpool hash function (full version: 10 rounds). Later Lamberger *et al.* extended the collision attack to 5 rounds [16]. After that, Sasaki published a (second) preimage attack on 5-round Whirlpool hash function in 2011 [25], and the complexity of his attack was improved by Wu *et al.* in 2012 [31]. Later Sasaki *et al.* extended the preimage attack to 6 rounds [27]. In addition to hash function attacks, several cryptanalysis results on the compression function of Whirlpool have also been published [16, 27], and particularly a distinguisher on the full compression function was found [16].

HMAC [2] was proposed by Bellare *et al.* in 1996. It has been standardized by ANSI, IETF, ISO and NIST, and widely deployed in SSL, TLS and IPsec. HMAC based on a hash function H takes a secret key K and a message M as input and is computed by

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

where ipad and opad are two different public constants. HMAC is always viewed as a single-key variant of NMAC [2]. NMAC based on a hash function H takes two keys; the inner key K_{in} and the outer key K_{out}, and a message M as input, and is computed by

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

where the function $H_{K_{in}}(\cdot)$ stands for the hash funtion H with its initial value replaced by K_{in}, and similarly for $H_{K_{out}}(\cdot)$. The internal states $F(IV, K \oplus \mathtt{opad})$ and $F(IV, K \oplus \mathtt{ipad})$ of HMAC is equivalent to the K_{out} and the K_{in} of NMAC respectively, where F is the compression function and IV is the public initial value of H. This paper refers $F(IV, K \oplus \mathtt{opad})$ and $F(IV, K \oplus \mathtt{ipad})$ to as the equivalent outer key and the equivalent inner key respectively. Note that if these two equivalent keys are recovered, the attacker will be able to forge any message, resulting in a universal forgery attack on HMAC.

Our contribution. We present universal forgery (*i.e.*, recover the two equivalent keys) and key recovery attacks on HMAC based on round-reduced Whirlpool, and a distinguishing-H attack on HMAC based on full Whirlpool. These attacks are also applicable to NMAC based on Whirlpool. All the results are summarized in Table 1. Interestingly, our attacks on the Whirlpool hash function in HMAC and NMAC setting reach attacked round numbers comparable to that in the public setting (even with respect to classical security notions; forgery and key recovery in MAC setting and collision and preimage attacks in public setting).

For HMAC and NMAC based on 5-round Whirlpool, we generate a structured collision on the first message block of the first call of hash function, which can be detected from the MAC output collisions and verified by the length extension property. For the structured collision, we know the differential path inside the block cipher $E_{K_{in}}$. Based on it, we apply a meet-in-the-middle attack to recover the value of K_{in}. After that, we apply two attacks. One is to recover the value of K_{out}, which results in a universal forgery attack on HMAC and a full-key recovery attack on NMAC. The attack of recovering K_{out} is similar with that of recovering K_{in}, except the procedure of finding target pairs. Instead of generating collisions as for recovering K_{in}, we will first recover the values of an intermediate chaining variable of the outer hash function, and then find a near collision on this intermediate chaining variable. The other attack is to recover the key of HMAC. Recall that $K_{in} = F(IV, K \oplus \texttt{ipad})$, recovering K from K_{in} is similar to inverting $F(IV, \cdot)$ to find a preimage of K_{in}. Thus we apply an attack similar with the splice-and-cut preimage attack to recover K from K_{in}. To our best knowledge, this is the *first* result of recovering the (original) key of HMAC, while previous results [11, 22, 23, 29] only succeeded in recovering the equivalent keys.

We investigate the extension by one more round, namely 6-round Whirlpool, and find an interesting observation. More precisely, K_{out} can be recovered if a value of an intermediate chaining variable in the first call of hash function is recovered or leaked. Differently from the above attacks on 5 rounds, the procedure is based on generating a multi-near-collision on an intermediate chaining variable of the outer hash function. After K_{out} is recovered, we apply two attacks. One is to recover K_{in}, which results in a universal forgery attack on HMAC and a full-key recovery attack on NMAC. The other attack is to recover the key of HMAC. From a high-level overview, our observation reduces the problem of breaking the classical security notions (with significant impacts) universal forgery and key recovery to the problem of breaking a weak security notion (usually with rather limited impacts) internal-state recovery for HMAC and NMAC based on 6-round Whirlpool. We stress that such a reduction is not trivial. As an example, an internal-state recovery attack was published on HMAC/NMAC-MD5 in the single-key setting back to 2009 [30], but no universal forgery or key recovery attack is published on HMAC/NMAC-MD5 in the single-key setting yet to our best knowledge. Moreover, very recently Leurent *et al.* find a generic single-key internal-state recovery attack on HMAC and NMAC [17]. Combing their attack with our observation, we get universal forgery and key recovery attacks on HMAC and NMAC based on 6-round Whirlpool.

We would like to point out that the above universal forgery and key recovery attacks on round-reduced Whirlpool are also applicable in other hash-based MAC setting. More precisely, we can attack LPMAC and secret-suffix MAC with 6-round Whirlpool and Envelop MAC with 5-round Whirlpool, all in the single-key setting.

Lastly, we find a distinguishing-H attack on HMAC and NMAC with full Whirlpool, which in fact has wide applications besides Whirlpool. Recall HMAC and NMAC make two calls of hash function, and the outer hash function takes the inner hash outputs as input messages. Thus the outer hash function always processes n bits long messages, where n is the bit size of hash digests. Note that usually

Table 1. Summarization of our results. These results are based on the minimization of max{data, time, memory}. More tradeoffs towards minimizing each parameter of data, time and memory are provided in the paper.

Our Result Summarization						
Attack Target	#Rounds	Attack mode	Complexity			Reference
			Time	Memory	Data	
HMAC-Whirlpool	5	universal forgery	2^{402}	2^{384}	2^{384}	Section 3
	5	key recovery	2^{448}	2^{377}	2^{321}	Section 3
	6	universal forgery	2^{451}	2^{448}	2^{384}	Section 4
	6	key recovery	2^{496}	2^{448}	2^{384}	Section 4
	10 (full)	distinguishing-H	2^{256}	2^{256}	2^{256}	Section 5
	10 (full)	distinguishing-H	2^{384}	2^{256}	2^{384}	[17]
NMAC-Whirlpool	5	key recovery	2^{402}	2^{384}	2^{384}	Section 3
	6	key recovery	2^{451}	2^{448}	2^{384}	Section 4
	10 (full)	distinguishing-H	2^{256}	2^{256}	2^{256}	Section 5
	10 (full)	distinguishing-H	2^{384}	2^{256}	2^{384}	[17]

Previous best results on Whirlpool hash function						
Whirlpool	5	collision attack	2^{120}	2^{64}	–	[16]
	6	preimage attack	2^{481}	2^{256}	–	[27]

the length and the value of the padding bits are solely determined by the bit size of an input message. Therefore it is possible that the last block of the outer hash function of HMAC and NMAC contains fully padding bits and thus is with a constant value, and indeed this is the case for HMAC- and NMAC-Whirlpool. Our distinguishing-H attack can be applied with a complexity $2^{n/2}$ (n is 512 for Whirlpool). Our distinguisher has two advantages compared with Leurent *et al.*'s generic attack [17]. One is that our queried messages have practical length. The other one is that the complexity of our attack is significantly lower as long as the specification of the n-bit hash function restricts the input message with a block length shorter than $2^{n/2}$. Our distinguishing-H attack on HMAC- and NMAC- Whirlpool has a complexity of 2^{256}, while Leurent *et al.*'s attack has a complexity of at least 2^{384}.

Note that we focus on HMAC-Whirlpool using a 512-bit key and producing 512-bit MAC outputs in this paper. One may doubt the large size of the key and the tag. We would like to point out that besides pure theoretical research interests, evaluating such an instance of HMAC-Whirlpool also has practical impacts. This is due to the fact that ever since HMAC was designed and standardized, it has been widely implemented beyond the mere MAC applications. For example, the above instance of HMAC-Whirlpool will be used in HMAC-based Extract-and-Expand Key Derivation Function (HKDF) [15] if one instantiates this protocol with Whirlpool hash function, providing that Whirlpool is a long-stand secure hash function and has been implemented in many cryptographic software library. Based on these facts, HMAC-Whirlpool may have more applications in industry in the future, and thus deserves a careful security evaluation from the cryptography community in advance.

In the rest of the paper, Section 2 gives the specifications. Section 3 presents our attacks on HMAC and NMAC with 5-round Whirlpool. Section 4 describes our results on one more round. Section 5 provides a distinguishing-H attack on HMAC and NMAC with full Whirlpool. Finally we give conclusion and open discussions in Section 6.

2 Specifications

2.1 Whirlpool hash function [24]

The Whirlpool hash function follows the Merkle-Damgård structure and produces 512-bit digests. The input message M is padded by a '1', a least number of '0's, and 256-bit representation of the original message length, such that the padded message becomes a multiple of 512 bits.

The padded message is divided into 512-bit blocks and used in the iteration of compression functions. The compression function F is constructed based on a block-cipher E in Miyaguchi-Preneel mode (MP mode), *i.e.*, $F(C, M) = E_C(M) \oplus C \oplus M$. Starting from a constant initial value $C_0 = IV$, the chaining value is updated for each of the message block $C_{i+1} = F(C_i, M_i)$. After all message blocks are processed, the final chaining value is used as the hash value.

The underlying block cipher uses an AES-like structure with an 8×8 byte matrix. The round function of the key schedule consists of four operations, *i.e.*,

$$K_{i+1} = \mathtt{AC} \circ \mathtt{MR} \circ \mathtt{SC} \circ \mathtt{SB}(K_i), \text{ for } i \in \{0, 1, \ldots, 9\}.$$

- SubBytes(SB): apply an Sbox to each byte.
- ShiftColumns(SC): cyclically rotate the j-th column downwards by j bytes.
- MixRows(MR): multiply the state by an 8×8 MDS matrix.
- AddRoundConstant(AC): XOR a 512-bit round constant to the key state.

We denote the key state after SB, after SC and after MR in the $(i + 1)$-th round of the key schedule by K_i^{SB}, K_i^{SC}, K_i^{MR} respectively.

The round function of the encryption is almost the same as the key schedule, except for the AddRoundKey(AK) operation, which XORs the key state to the data state, *i.e.*, the initial state is the XOR sum of the whitening key and the plaintext $S_0 = K_0 \oplus M$ and

$$S_{i+1} = \mathtt{AK} \circ \mathtt{MR} \circ \mathtt{SC} \circ \mathtt{SB}(S_i), \text{ for } i \in \{0, 1, \ldots, 9\}.$$

The final state S_{10} is used as the ciphertext. We denote the state after SB, after SC and after MR in the $(i+1)$-th round of the round encryption function by S_i^{SB}, S_i^{SC} and S_i^{MR} respectively.

2.2 HMAC and NMAC [2]

NMAC replaces the initial vector (IV) of a hash function H by a secret key K to produce a keyed hash function H_K. NMAC uses two secret key K_{in} and K_{out} and is defined by

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

K_{in} and K_{out} are usually referred to as the inner and the outer keys. Correspondingly $H_{K_{in}}$ and $H_{K_{out}}$ are referred to as the inner and the outer hash functions. HMAC is a single-key variant of NMAC. Denote the compression function by F.

$$\mathtt{HMAC}_K(M) = \mathtt{NMAC}_{F(IV,K\oplus\mathtt{ipad}),F(IV,K\oplus\mathtt{opad})}(M).$$

3 Attacks of HMAC and NMAC based on 5-round Whirlpool

In this section, we use one block long messages M to present our attack. Fig. 2 shows how HMAC/NMAC-Whirlpool processes M. Note that both M and T'' are one full block long, and thus an extra padding block P is appended in both the two calls of the hash function.

The attack starts with recovering value of (equivalent) K_{in}. We generate a structured collision on the internal state T'. Then for the collision, we get the differential path inside $E_{K_{in}}$, and recover some internal value of $E_{K_{in}}$ by a meet-in-the-middle (MitM) attack approach. Finally K_{in} is derived by a simple backward computations. Once K_{in} is recovered, we have two directions: 1) recover the value of (equivalent) K_{out} to amount a universal forgery for HMAC or to amount a full-key recovery for NMAC, and 2) recover the key of HMAC.

For the K_{out} recovery, note that T'' is public to the attacker now since K_{in} is recovered. We firstly derive the values of T''' with a technique similar to [26], and then obtain the values of $E_{K_{out}} \oplus K_{out}$: $T'' \oplus T'''$. Given that K_{out} has no difference, we search for a pair of messages that satisfies a pre-determined difference constraint on the outputs of $E_{K_{out}}$, and get the inside differential path. Finally we recover an internal value of $E_{K_{out}}$, and backwards compute the value of K_{out}.

For the key recovery of K, from $K_{in} = F(IV, K \oplus \mathtt{ipad})$, we observe that $K \oplus \mathtt{ipad}$ is a preimage of K_{in} regarding the Whirlpool compression function with a fixed chaining value $F(IV, \cdot)$. Note that the problem of inverting the compression function of Whirlpool has already been solved in [31] and [27] with splice-and-cut MitM approach. We use a similar approach to recover the value of $K \oplus \mathtt{ipad}$ and then derive the value of K.

Moreover, we provide time-memory-data tradeoffs for recovering K_{in} and K_{out}.

3.1 How to recover (equivalent) K_{in}

In this section, we demonstrate the K_{in} recovery attack with optimizing its complexity for the key recovery of HMAC, and introduce the time-memory-data tradeoff in the next section.

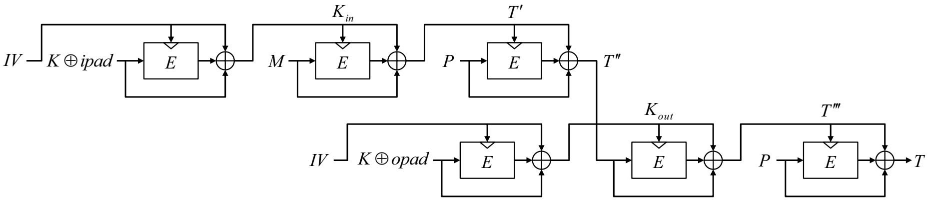

Fig. 2. `HMAC`/`NMAC` based on `Whirlpool`

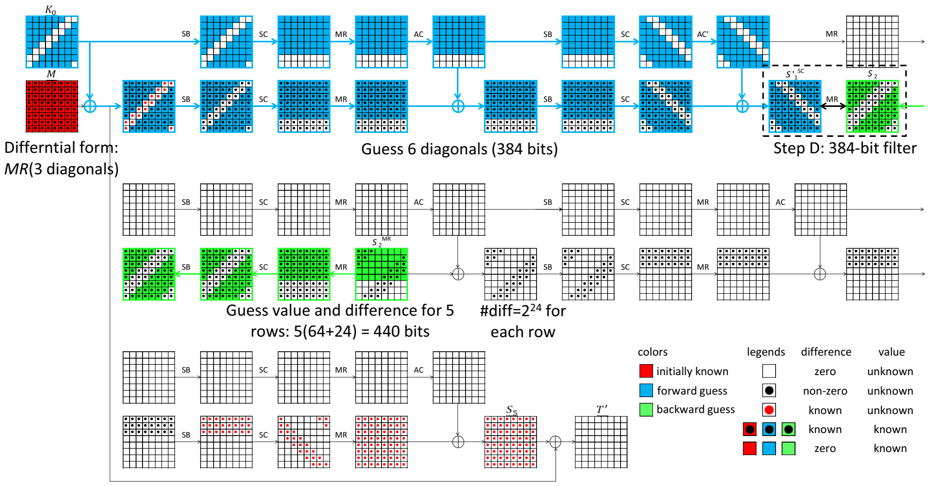

Fig. 3. Differential path for recovering K_{in} for `HMAC` and `NMAC` with 5-round `Whirlpool`

Our attack is based on a 5-round differential path of the compression function, which is shown in Fig. 3. Each cell in this figure stands for a byte of the key or the state. Blank cells are non-active and cells with a dot inside are active. If the value of a byte is unknown, the cell is in white color. Red bytes are initially known from the message, tag or the recovered chaining value. Blue and green bytes are the guessed bytes in the forward and backward directions of the MitM step. Moreover, some round functions are illustrated in equivalent expressions in this figure. The new operation $\texttt{AC}'$ XORs the constant $\texttt{MR}^{-1}(RC_i)$ to the key state, where RC_i is used in the original `AC` operation, and it implies $\texttt{AC} \circ \texttt{MR} = \texttt{MR} \circ \texttt{AC}'$.

Produce a structured collision on T'. We use a structure of chosen messages in which any two messages satisfy a constraint of the differential form in Fig. 3. First we choose a set of 2^{192} values $\{M_1, M_2, \ldots, M_{2^{192}}\}$ such that the value of three specific rows of the messages take all possible 2^{192} values and all other bytes are chosen as constants. The positions of the three active rows are the top three rows in Fig. 3. Then, update the set by $M_i \leftarrow \texttt{MR} \circ \texttt{SC}(M_i)$ for $i = 1, 2, \ldots, 2^{192}$. This requires about 2^{192} computations. Note that for any two distinct indexes i_1

and i_2, $\texttt{SC}^{-1} \circ \texttt{MR}^{-1}(M_{i_1} \oplus M_{i_2})$ has three active rows in the pre-specified positions. Query the messages and obtain the corresponding tags $T_i = \texttt{MAC}(K, M_i)$, for $i = 1, 2, \ldots, 2^{192}$. Check if there is a collision of the tags. If a collision is found, we need to verify if it collides on T' by the length extension attack (*i.e.*, append a random message block M to each of the colliding messages M_{i_1} and M_{i_2}, and query $M_{i_1} \| M$ and $M_{i_2} \| M$ to see whether their tags collide). For a collision on T', it is ensured that the output difference of $E_{K_{in}}$ converted by $\texttt{SC}^{-1} \circ \texttt{MR}^{-1}$ has three active rows.

For a structure of 2^{192} messages generated by applying $\texttt{MR} \circ \texttt{SC}$ for each, we query them to MAC, store the corresponding tags and search for a collision. So it requires 2^{192} queries, 2^{192} computations, and 2^{192} memory. For one structure, we can make $\binom{2^{192}}{2} = 2^{383}$ pairs. After repeating the process for 2^{129} structures with different chosen constants, one collision is expected. The total number of queries is $2^{192+129} = 2^{321}$, the computational complexity is 2^{321} and the required memory is 2^{192}.

Recover K_{in}. Recall $T' = F(K_{in}, M) = E_{K_{in}}(M) \oplus M \oplus K_{in}$. For an inner collision on T', we know $\triangle T' = 0$. In the single-key attacks, the difference of K_{in} is also zero: $\triangle K_{in} = 0$. Thus the difference of the output of the block cipher can be computed as $\triangle E_{K_{in}}(M) = \triangle T' \oplus \triangle M \oplus \triangle K_{in} = \triangle M$. So we get $\triangle S_5 = \triangle M$, and thus $\texttt{SC}^{-1} \circ \texttt{MR}^{-1}(\triangle S_5)$ has three active rows. It ensures that the number of differences at each row of S_2^{MR} is at most 2^{24}. Now we describe the attack step by step.

Step A. Guessing in the forward direction
Guess the values of m diagonals of K_{in} (2^{64m} values) which are marked in blue, as in Fig. 3. Then we can determine the value of corresponding m diagonals in ${S'}_1^{SC}$. Now there are m known diagonals on the left side of the matching point - the MR operation in the second round. All the candidates are stored in a lookup table T_1.

Step B. Guessing in the backward direction
Guess the values and differences of n rows of S_2^{MR} ($2^{(64+24)n}$ candidates) which are marked in green, as in Fig. 3. Then we can determine the value of corresponding n (reverse) diagonals in S_2. Now there are n known diagonals on the right side of the matching point. All the candidates are stored in another lookup table T_2.

Step C. MitM matching across the MR operation
The technique of matching across an MDS transformation is already proposed and well-discussed in [25, 31, 27]. Here we directly give the result. For a 64-byte state, the bit size of the matching point is calculated as $64(m+n-8)$, where m and n are the number of known diagonals in both sides. Because we can match both of the value and difference on a 64-byte state, the bit size of the matching point is $128(m+n-8)$. Therefore, the number of expected matches between T_1 and T_2 is $2^{64m+(64+24)n-128(m+n-8)} = 2^{-64m-40n+1024}$. Note that the matching candidate is a pair of $({S'}_1^{SC}, S_2)$ where all bytes are fully determined. Then, the corresponding ${K'}_1^{AC}$ is also fully determined.

We use a pre-computation of complexity 2^{65} to build a table of size 2^{65}, which is used for $({S'}_1^{SC}, S_2)$ to determine the remaining two diagonals of the corresponding K_{in} by just a table lookup. More precisely, for all values of each unguessed diagonal of K_{in}, compute the corresponding diagonal values in ${S'}_1^{SC}$, and store them in a lookup table. The number of remaining candidates is also the number of suggested keys. The correctness of each suggested keys can be verified by the differential path from S_3 to S_5.

The total complexity of the attack is

$$2^{64m} + 2^{(64+24)n} + 2^{-64m-40n+1024}.$$

When $m = 6$ and $n = 5$, we get the complexity of about $2^{384} + 2^{440} + 2^{440} \approx 2^{441}$ computations. The sizes of T_1 and T_2 are 2^{384} and 2^{440} respectively. Since we only need to store one of them and leave the calculations of other direction "on the fly", the memory requirement is 2^{384}. Taking into account the phase to find the inner collision, the total time complexity for recovering K_{in} is 2^{441} time and 2^{384} memory, along with 2^{321} chosen queries. Recall that we chose the attack parameters by considering that the original key recovery attack on HMAC will require 2^{448} computations as we later show in Section 3.4. We balanced the time complexity and then reduced the memory and queries as much as possible.

3.2 Time-Memory-Data Tradeoff for K_{in} Recovery

For the differential path in Fig. 3, the number of active rows does not have to be three. Indeed, this derives a tradeoff between data (the number of queries) and time-memory. Intuitively, the more data we use, the more restricted differential path we can satisfy and thus time and memory can be smaller. On the other hand, data can be minimized by spending more time and memory. Let r be the number of active rows in Fig. 3. For a single structure, $\binom{2^{64r}}{2} = 2^{128r-1}$ pairs can be constructed with 2^{64r}queries. In the end, a collision can be found with $2^{513-64r}$ queries.

Then, the MitM phase is performed. The time complexity for the forward computation does not change, which is 2^{64m}, while the complexity for the backward computation is dependent on r, which is $2^{(64+8r)n}$. We can further introduce the tradeoff between time and memory, where their product takes a constant value. For simplicity, let us assume that $2^{64m} < 2^{(64+8r)n}$. The simple method computes the forward candidates with 2^{64m} computations and stores them. Then, the backward candidates are computed with $2^{(64+8r)n}$. Hence, the time is $2^{(64+8r)n}$ and the memory is 2^{64m}. Here, we divide the free bits for the forward computation into two parts; $64m - t$ and t. An attacker firstly guesses the value of t bits, and for each guess, computes the 2^{64m-t} forward candidates and stores them in a table with 2^{64m-t} entries. The backward computation does not change. Finally, the attack is iterated for 2^t guesses. In the end, the memory complexity becomes 2^{64m-t} while the time complexity becomes $2^{(64+8r)n+t}$ computations.

Let us demonstrate the impact of the time-memory-data tradeoff. In section 4.1, we aimed to achieve the time complexity of 2^{448}, and chose the parameter $(r, m, n, t) = (3, 6, 5, 0)$ which resulted in (data, time, memory) $= (2^{321}, 2^{441}, 2^{384})$. By choosing parameters $(r, m, n, t) = (3, 6, 5, 7)$, memory can be saved by 7 bits, *i.e.*, (data, time, memory)$= (2^{321}, 2^{448}, 2^{377})$. Then let us consider the optimization from different aspects. First, we minimize the value max{data, time, memory}. We should choose $(r, m, n, t) = (2, 6, 5, 0)$, which results in (data, time, memory)$= (2^{384}, 2^{400}, 2^{384})$. Next, we try to minimize each of time, data, and memory complexities. If we minimize the time complexity, we should choose $(r, m, n, t) = (1, 6, 5, 0)$, which results in (data, time, memory)$= (2^{448}, 2^{384}, 2^{360})$. If we minimize the data complexity, we should choose $(r, m, n, t) = (4, 7, 5, t)$ which results in (data, time, memory)$= (2^{257}, 2^{480+t}, 2^{448-t})$. If we minimize the memory complexity, we should choose $(r, m, n, t) = (1, 6, 5, 144)$ which results in (data, time, memory)$= (2^{449}, 2^{504}, 2^{240})$.

3.3 How to recover K_{out}

With the knowledge of K_{in}, we can calculate the value of T'' for any M at offline (refer to Fig. 2). Moreover, we can recover the value of T''' using a technique similar to [26]. Thus we are able to get the output value of $E_{K_{out}} \oplus K_{out}$: $T'' \oplus T'''$. For a pair of outputs of $E_{K_{out}} \oplus K_{out}$ that has a difference satisfying the constraint on the output difference of $E_{K_{in}}$ in Fig. 3, more precisely $\mathtt{SC}^{-1} \circ \mathtt{MR}^{-1}(\Delta(T'' \oplus T'''))$ has r active rows, the exactly same procedure of recovering K_{in} described in Section 3.1 can be applied to recover K_{out} in a straight-forward way. This section mainly describes the procedure of finding such a pair. Moreover, we provide a time-memory-data tradeoff for recovering K_{out}.

It is interesting to point out the difference for finding a target pair of recovering K_{in} and that of recovering K_{out}. For recovering K_{in}, we can freely choose the input M, but cannot derive the output differences of $E_{K_{in}}$ unless a collision occurs on the compression function. For recovering K_{out}, we cannot control the input T'', but can compute the output differences of $E_{K_{out}}$ easily since we know the values of both T'' and T'''.

Produce $(8-r)$-row near collision on $\mathtt{SC}^{-1} \circ \mathtt{MR}^{-1}(T'' \oplus T''')$. We need to recover the value of T''', which is as follows. Firstly, choose 2^x different random values X_i, calculate $Y_i = F(X_i, P)$ and store (X_i, Y_i) in a lookup table T_1 at offline. Secondly, choose 2^y different random values of M_i, query them to $\mathtt{MAC}$, obtain $Z_i = \mathtt{MAC}(K, M_i)$ and store (M_i, Z_i) in another lookup table T_2. Finally we match Y_i in T_1 and Z_j in T_2, and get $2^{x+y-511}$ matches on average. For each match, the internal state T''' of M_j is equal to X_i with a probability 1/2. We stress that in fact we need to store only one of T_1 and T_2, and generate the other on the fly.

Next, we continue to search for a target pair. For each match of Y_i and Z_j, we compute the value of T'' of M_j, then compute $W = \mathtt{SC}^{-1} \circ \mathtt{MR}^{-1}(T'' \oplus X_i)$, and store them in a lookup Table T_3 to find $(8-r)$-row near collisions on W.

Recall the recovered value of T''' of a message is correct with a probability of 1/2. Thus we need to generate 4 near collisions to ensure that one is a target pair. It implies $2^{2(x+y-511)-1} = 4 \times 2^{64(8-r)}$, and thus $2x + 2y + 64r = 1537$.

In total, the data complexity is 2^y queries, the time complexity is $2^x + 2^{x+y-511}$, and the memory requirement is $\min\{2^x, 2^y\}$.

Time-memory-data tradeoff. The attack contains two tradeoffs. The first one is for finding a target pair, which is described above. The second one is for MitM phase, which is described in Section 3.2. Note that the MitM has to be applied to all the 4 near collisions, and so the time complexity of the tradeoff for MitM phase is increased by 4 times. Both tradeoffs depend on the parameter r, and thus we first determine the value of r, and analyze the two tradeoffs together. We provide the parameters that minimize the time complexity, the data complexity, or the memory complexity. Note that recovering K_{out} needs to recover K_{in} first, and so we should also take the tradeoff results on recovering K_{in} into account. Let us minimize the value $\max\{\text{data}, \text{time}, \text{memory}\}$. Considering that the same MitM procedure of recovering K_{in} is used for recovering K_{out}, we just need to minimize that of recovering K_{in}, and obtain that $(\text{data}, \text{time}, \text{memory}) = (2^{384}, 2^{402}, 2^{384})$. If we minimize the time complexity, we should choose parameters $(r, m, n, t) = (1, 6, 5, 0)$ for recovering K_{in} and $(x, y, r, m, n, t) = (360, 397, 1, 6, 5, 0)$ for recovering K_{out}, which results in $(\text{data}, \text{time}, \text{memory}) = (2^{448}, 2^{386}, 2^{360})$. If we minimize the data complexity, we should choose parameters $(r, m, n, t) = (4, 7, 5, t)$ for recovering K_{in} and $(x, y, r, m, n, t) = (448, 225, 3, 6, 5, 0)$ for recovering K_{out}, which results in $(\text{data}, \text{time}, \text{memory}) = (2^{257}, 2^{480+t}, 2^{448-t})$. If we minimize the memory complexity, then the time and data are dominated by the K_{in} recovery, and thus $(\text{data}, \text{time}, \text{memory}) = (2^{449}, 2^{504}, 2^{240})$ by choosing the parameters given in Section 3.2.

3.4 Key recovery for HMAC

As previously mentioned, we will recover the key of HMAC based on the splice-and-cut preimage attacks on the compression function with a fixed input chaining variable $F(IV, \cdot)$.

The attack model for the preimage attack on hash functions and the one for the key recovery attack on HMAC are slightly different. For a given hash value, there are two possibilities: 1) there exists one or more preimages; 2) no preimage exists. For the first case, the aim of the attacker is to find *any one* of the preimages, instead of all of them. The second case may occur when the size of input is restricted. In our sub-problem, *i.e.*, for a compression function with fixed chaining value, the sizes of the input message and the hash digest are the same. Thus for a random output, the probability that no preimage exists is not negligible: about e^{-1}.

For a key recovery attack, the solution (the secret key) always exists. However, the attacker has to go over *all* possible preimages to ensure that the correct key is covered. In the process of the MitM attack, sometimes there is some entropy loss in the initial structure, *i.e.*, the attacker only looks for the preimage in a subspace. When the size of the input space is bigger than the output space, a preimage attack is still possible with entropy loss. If such a preimage attack is used for key recovery, the real key could be missed.

In the preimage attacks of [31] and [27], no entropy is lost and all the possible values of the state can be covered. Thus the key recovery attack based on this preimage attack can always succeed. The complexity is 2^{448} time and 2^{64} memory.

Recall the discussion in Section 3.1, where K_{in} is recovered with (data, time, memory) $= (2^{321}, 2^{448}, 2^{377})$. Together with the preimage attack on the compression function, the original key for HMAC with 5-round `Whirlpool` is recovered with $(\text{data}, \text{time}, \text{memory}) = (2^{321}, 2^{448}, 2^{377})$.

3.5 Summary

In short, we have solved three sub-problems: (1) Recover K_{in} with 2^{384} chosen queries, 2^{400} time and 2^{384} memory. Then with the knowledge of K_{in}, we can solve another two sub-problems:(2) Recover K_{out} with 2^{303} known message-tag pairs, 2^{402} time and 2^{384} memory; and (3) Recover the key of `HMAC` from K_{in} with 2^{448} time and 2^{64} memory. The time-memory-data tradeoff exists in (1), and we can optimize its complexity depending on the final goal; (2) or (3).

Combining (1) and (2) with optimized trade-off, we have a key recovery attack on `NMAC` and a universal forgery attack on `HMAC` with 2^{384} chosen queries, 2^{402} time, and 2^{384} memory. Combining (1) and (3) with optimized trade-off, we have a key recovery attack on `HMAC` with 2^{321} chosen queries, 2^{448} time, and 2^{377} memory.

4 Analysis of `HMAC` and `NMAC` based on 6-round `Whirlpool`

This section presents how to extend an attack of recovering an intermediate chaining variable of the inner hash function to universal forgery and key recovery attacks for `HMAC` and `NMAC` with 6-round `Whirlpool`. Note that a generic single-key attack of recovering such an intermediate chaining variable for `HMAC` and `NMAC` has been published by Leurent *et al.* [17]. It takes around a complexity of 2^{384} blocks for all queries to recover an internal state value of a short message, e.g. one block long. Thus combining their results with our analysis, we get universal forgery and key recovery attacks on `HMAC` with 6-round `Whirlpool`.

In the rest of this section, we denote by M_1 the message whose intermediate chaining value is recovered by the attacker. We start with recovering K_{out}, which is depicted in Fig. 4.

Produce a 3-near-collision on $\texttt{MR}^{-1}(T'' \oplus T''')$. We append random messages M' to M_1, and query them to `MAC`. Note that we are able to compute their values of T'' at offline. We recover the values of T''' for those messages

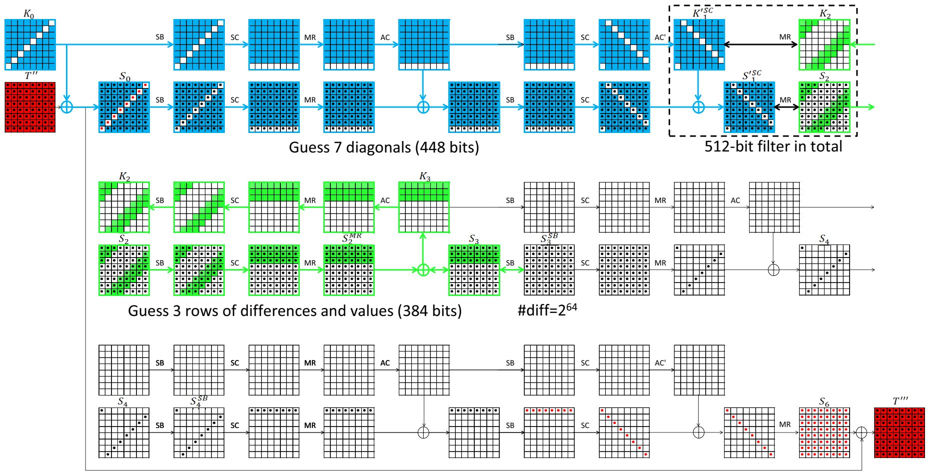

Fig. 4. How to recover K_{out} for `HMAC` and `NMAC` with 6-round `Whirlpool`

in the same way as we did for 5-round `Whirlpool`. After that, we compute $W = \texttt{MR}^{-1}(T'' \oplus T''')$, and search three messages that all collide on specific 56 bytes of W as shown in Fig. 4. We call such three messages 3-near-collision.

With 2^x online queries and 2^y offline computations, $2^{x+y-511}$ values of T''' are recovered, and the same number of W are collected. Note that around $\sqrt[3]{3!} \cdot 2^{\frac{2}{3}448} \approx 2^{300}$ values of W are necessary to find a target 3-near-collision [10]. Moreover, we need to generate 8 such 3-near-collisions to ensure one is indeed our target, since each value of recovered T''' is correct with a probability 1/2. So we get $2^{x+y-511} = 2^{303}$, which implies that $x + y = 814$.

Recover K_{out}. A pair of messages from a 3-near-collision follows the differential path in Fig. 4 such that only one (reversed) diagonal of S_4 is active. Thus the number of possible differences in S_3^{SB} is 2^{64}. Denote three messages of a 3-near-collision as m_1, m_2 and m_3. Denote the values of the states ${S'}_1^{SC}$ and S_2 as L_i and R_i for m_i. We will apply the meet-in-the-middle attack two times, one for the pair (m_1, m_2) and the other for the pair (m_2, m_3).

Step A. Guessing in the forward direction

Guess the values of m diagonals of K_0 as shown in Fig. 4 (2^{64m} values, marked in blue) and determine the value of corresponding m diagonals in ${K'}_1^{SC}$ and ${S'}_1^{SC}$.

Step B. Guessing in the backward direction

Guess the values and differences of n diagonals of S_2 (2^{128n} values) which are marked in green. Then we can determine the value of corresponding n rows in S_2^{MR}. After the injection of K_3, we only know the difference in S_3. Since the number of possible differences of S_3^{SB} is only 2^{64}. According to rebound attack, we expect 2^{64} solutions for each guess of S_2. XOR 2^{64} values of S_3

and the guessed value of S_2^{MR}, and obtain 2^{64} values for the top n rows of K_3 and n diagonals of K_2. In total, the number of candidates on the right side of the MitM part is $2^{128n+64}$.

Step C. MitM matching across MR on both the key and the state
For the differential path between m_1 and m_2, the value and difference of S_2 are in fact R_1 and $R_1 \oplus R_2$. Once we have matched the value and difference of the state, *i.e.*, $\texttt{MR}(L_1) = R_1$ and $\texttt{MR}(L_1 \oplus L_2) = R_1 \oplus R_2$, it is equivalent to match both the values $\texttt{MR}(L_1) = R_1$ and $\texttt{MR}(L_2) = R_2$. For the second differential path between m_2 and m_3, we only need to match another state $\texttt{MR}(L_3) = R_3$, since $\texttt{MR}(L_2) = R_2$ is already fulfilled. We come to an observation that the size of the matching point (filter size) is actually $(1 + 3) \times 64 \times (m + n - 8)$ bits, *i.e.*, one from the key, three from the 3-near-collision. The expected number of matches (suggested keys) is $2^{64m+128n+64-256(m+n-8)} = 2^{2112-192m-128n}$.

The overall complexity to recover K_{out} is

$$2^{64m} + 2^{128n+64} + 2^{2112-192m-128n}.$$

When $m = 7$ and $n = 3$, the complexity is 2^{448} time and memory.

Note that the above procedure will be applied by 8 times. Thus the time complexity is 2^{451}. By setting $y = 451$, we get $x = 363$, and thus the data complexity is dominated by the recovery of an intermediate chaining variable of the inner hash function [17], namely 2^{384}.

Universal forgery and key recovery. After K_{out} is recovered, almost the same procedure can be applied to recover K_{in}. So we get universal forgery on `HMAC` and full-key recovery on `NMAC`. Note that for recovering K_{in}, it is easy to verify whether a obtained 3-near-collision is our target. Thus the total complexity is dominated by recovering K_{out}, and we get (data, time, memory)=$(2^{384}, 2^{451}, 2^{448})$. Moreover, we apply the splice-and-cut preimage attack to recover K from K_{out} according to [27], which takes a time complexity of 2^{496} and a memory requirement of 2^{64}. Thus the total complexity of recovering K is (data, time, memory)=$(2^{384}, 2^{496}, 2^{448})$.

5 Distinguishing-H Attack on Full `HMAC/NMAC-Whirlpool`

In this section, we present a distinguishing-H attack on `HMAC-Whirlpool`, which is also applicable to `NMAC-Whirlpool` in a straightforward way. First, recall the definition of distinguishing-H [14]. A distinguisher `D` is to identify an oracle being either `HMAC-Whirlpool` or another primitive built by replacing the compression function F of `HMAC-Whirlpool` to a random function R with the same domain and range. For a hash function with n-bit digests, it is believed that a generic distinguishing-H attack requires 2^n complexity if the hash function is ideal.

We observe that during the computation of the outer `Whirlpool` in `HMAC-Whirlpool`, the last message block is always a constant denoted as P, more

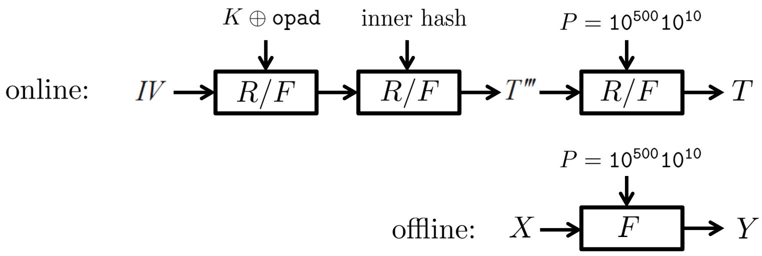

Fig. 5. Distinguishing-H attack on `HMAC-Whirlpool`

precisely $P = \texttt{1}\texttt{0}^{500}\texttt{1}\texttt{0}^{10}$ where 0^l means l consecutive 0s. This is because of the equal size of message block and hash digest and the padding rule of `Whirlpool`. The input messages to the outer `Whirlpool` consist of one block of $K \oplus \texttt{opad}$ and one block of the inner `Whirlpool` digest, and thus are always two full blocks long (namely 1024 bits), which are padded with one more block. Note that the padded block P, which is the last message block of the outer `Whirlpool`, is solely determined by the bit length of the input messages, and thus is always a constant. Based on the observation, we launch a distinguishing-H attack.

We first explain the overview of the attack. In the online phase, query random messages M to the oracle, and receive tag values T. In the offline phase, choose random values X (this simulates the value of T'''), and compute $Y = F(X, P)$. As depicted in Fig. 5, if the compression function of the oracle is F, two events lead to the occurrence of $Y = T$: one is $X = T'''$; and the other is $F(X, P) = F(T''', P)$ under $X \neq T'''$. If the compression function of the oracle is R, only one event leads to the occurrence of $Y = T$: $F(X, P) = R(T''', P)$. Therefore, the probability of the event $Y = T$ in the former case is (roughly) twice higher than that in the latter case. Thus, by counting the number of occurrence of $Y = T$, the compression function being either F or R can be distinguished. A detailed attack procedure is described below.

Online phase. Send 2^{256} different messages M, which are one block long after padding, to the oracle. Receive the responses T and store them.

Offline phase. Choose a random value as X, and compute $Y = F(X, P)$. Match Y to the set of Ts stored in the online phase. If a match is found, terminate the procedure, and output 1. Otherwise, choose another random value of X and repeat the procedure. After 2^{256} trials, if no match is found, terminate the procedure and output 0.

The complexity is 2^{256} online queries and 2^{256} offline computations. The memory cost is 2^{256} tag values. Next we evaluate the advantage of the distinguisher. Denote by $\mathtt{D}^F$ the case $\mathtt{D}$ interacts with `HMAC-Whirlpool`, and by $\mathtt{D}^R$ the other case. The advantage of the distinguisher $\mathrm{Adv}_{\mathtt{D}}^{\mathtt{Ind-H}}$ is defined as

$$\mathrm{Adv}_{\mathtt{D}}^{\mathtt{Ind-H}} := |\Pr[\mathtt{D}^F = 1] - \Pr[\mathtt{D}^R = 1]|.$$

In the case of $\mathtt{D}^F$, the probability of $X = T'''$ is $1-(1-2^{-512})^{2^{512}} \approx 1-1/e \approx 0.63$ since there are 2^{512} pairs of (X, T'''). The probability of $Y = T$ and $X \neq T'''$ is $(1 - 1/e) \times 1/e \approx 0.23$. Therefore, $\Pr[\mathtt{D}^F = 1]$ is 0.86 $(= 0.63 + 0.23)$. In the other case, $\Pr[\mathtt{D}^R = 1]$ is 0.63 by a similar evaluation. Overall, the advantage of the distinguisher is 0.23 $(= 0.86 - 0.63)$.

Note that a trivial Data-Time tradeoff exists with the same advantage, Data $\times$ Time $= 2^{512}$.

Remarks on applications. We emphasize that the above distinguishing-H attack has wide applications besides `Whirlpool`. For example, there are 11 out of 12 collision resistance PGV modes [21] including well-known Matyas-Meyer-Oseas mode and Miyaguchi-Preneel mode such that the chaining variable and the message block have equal bit size due to either the feed-forward or the feed-backward operations. If a hash function `HF` is built by iterating one of those PGV compression function schemes in the popular (strengthened) Merkle-Damgård domain extension scheme, the last message block of the input messages to the outer `HF` in `HMAC` or `NMAC` setting is always a constant, and thus the above distinguishing-H attack is applicable.

6 Conclusion and Open Discussions

In conclusion, we presented the first forgery and key recovery attacks against `HMAC` and `NMAC` based on the `Whirlpool` hash function reduced to 5 out of 10 rounds in single key setting, and 6 rounds in related-key setting. In addition to `HMAC` and `NMAC`, our attacks apply to other `MAC`s based on reduced `Whirlpool`, such as `LPMAC`, `secret-suffix MAC` and `Envelop MAC`. We also gave a distinguishing-H attack against the full `HMAC`- and `NMAC-Whirlpool`.

As open discussion, it is interesting to see if the techniques presented in this paper are useful to analysis of other `AES`-like hash functions in hash-based `MAC` setting. First let us have a closer look at our analysis of the underlying `AES`-like block cipher in a hash function. One main and crucial strategy is restricting the differences to appear only in the encryption process and thus keep the key schedule process identical between the pair messages. For example, `Whirlpool` uses Miyaguchi-Preneel scheme $E_C(M) \oplus M \oplus C$ (notations follows Section 2), and the differences is introduced only by M. Recall through our analysis, C is kept the same during finding target message pairs. The main reason of this strategy is that a difference introduced from the keys propagates in both the key schedule and the encryption process, which usually makes it harder to analyze. For example, in our analysis on `HMAC-Whirlpool`, we need to derive the differential path in the encryption process, which becomes much harder when differences also propagate in the key schedule. Moreover, as briefly explained in

Section 1, differently from that in CBC MAC setting, one cannot derive a difference on intermediate hash variable ΔC except $\Delta C = 0$. Thus the difference has to be introduced from M. After an investigation on proven secure PGV schemes [21], we find that our analysis approach is applicable to other three schemes besides Miyaguchi-Preneel scheme: $E_C(M) \oplus M$ (well known as Matyas-Meyer-Oseas scheme), $E_C(C \oplus M) \oplus M$ and $E_C(C \oplus M) \oplus C \oplus M$.

It is also interesting to see if the strategies proposed to analyze MD4-like hash functions (designed in a framework differently from AES) can be applied to AES-like hash functions from a high-spirit level, in hash-based MAC setting. There are two strategies to analyze MD4-family hash function in hash-based MAC setting to the best of our knowledge. The first one was proposed by Contini and Yin [5]. Their strategy heavily relies on one design character of MD4-like hash function: a message block is splitted into words, and these words are injected into the hash process sequentially. More precisely, an attacker can fix the beginning message words that have been ensured to satisfy the first steps of differential path, and randomize the other message words. *Unfortunately*, this strategy seems not promising to be applied to AES-like hash functions, because the latter injects the whole message block into the hash process at the same time, and moreover a byte difference propagates to the whole state very quickly due to the wide trail design of AES. The other strategy was proposed by Wang *et al.* [30]. Their strategy uses two message blocks and each block have differences. Firstly they generate a high-probability differential path on the second compression function $(\Delta C, \Delta M) \rightarrow \Delta C' = 0$, where C' is the output of the second compression function. Secondly they randomize the first message block to generate pairs of the compression function outputs that can satisfy ΔC, and each such pair can be obtained by a birthday bound complexity. Finally these pairs will be filtered out using the high-probability differential path on the second compression function, and exploited to amount further attacks. *Interestingly*, this strategy seems applicable to AES-like hash functions in MAC setting. One may build a high-probability related-key differential path on an AES-like compression function, *e.g.*, using the local collisions between the key schedule and the encryption process functions which has been found on AES [3] and on Whirlpool [27]. If it is achieved, then Wang *et al.*'s strategy seems to be applicable. Note that previous constraint $\Delta C = 0$ is now removed, and thus this strategy has a potentiality to be applied to more PGV schemes such as $E_M(C) \oplus C$ (well known as Davies-Meyer scheme).

As our result is the first step in this research topic, we expect that future works will provide deeper understanding of the security of AES-like hash functions in MAC setting.

Acknowledgements

We would like to thank Jiqiang Lu, and anonymous reviewers for their helpful comments. This research was initially started from a discussion at the second Asian Workshop on Symmetric Key Cryptography (ASK 2012). We would like to thank the organizers of ASK12. Jian Guo, Lei Wang and Shuang Wu are supported by the Singapore National Research Foundation Fellowship 2012 (NRF-NRFF2012-06).

References

1. ISO/IEC 9797-1. Information Technology-security techniques-data integrity mechanism using a cryptographic check function employing a block cipher algorithm. International Organizatoin for Standards.
2. Mihir Bellare, Ran Canetti, and Hugo Krawczyk. Keying Hash Functions for Message Authentication. In Neal Koblitz, editor, *CRYPTO*, volume 1109 of *LNCS*, pages 1–15. Springer, 1996.
3. Alex Biryukov, Dmitry Khovratovich, and Ivica Nikolic. Distinguisher and Related-Key Attack on the Full AES-256. In Shai Halevi, editor, *CRYPTO*, volume 5677 of *Lecture Notes in Computer Science*, pages 231–249. Springer, 2009.
4. Charles Bouillaguet, Patrick Derbez, and Pierre-Alain Fouque. Automatic Search of Attacks on Round-Reduced AES and Applications. In Phillip Rogaway, editor, *CRYPTO*, volume 6841 of *LNCS*, pages 169–187. Springer, 2011.
5. 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.
6. Joan Daemen and Vincent Rijmen. *The Design of Rijndael: AES - The Advanced Encryption Standard*. Springer, 2002.
7. Joan Daemen and Vincent Rijmen. A New MAC Construction ALRED and a Specific Instance ALPHA-MAC. In Henri Gilbert and Helena Handschuh, editors, *FSE*, volume 3557 of *LNCS*, pages 1–17. Springer, 2005.
8. Joan Daemen and Vincent Rijmen. The Pelican MAC Function. *IACR Cryptology ePrint Archive*, 2005:88, 2005.
9. Orr Dunkelman, Nathan Keller, and Adi Shamir. ALRED Blues: New Attacks on AES-Based MAC's. *IACR Cryptology ePrint Archive*, 2011:95, 2011.
10. William Feller. *An introduction to probability theory and its applications*, volume 1. Wiley, New York, 3rd edition, 1967.
11. 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 *LNCS*, pages 13–30. Springer, 2007.
12. Jianyong Huang, Jennifer Seberry, and Willy Susilo. On the Internal Structure of Alpha-MAC. In Phong Q. Nguyen, editor, *VIETCRYPT*, volume 4341 of *LNCS*, pages 271–285. Springer, 2006.
13. Jianyong Huang, Jennifer Seberry, and Willy Susilo. A five-round algebraic property of AES and its application to the ALPHA-MAC. *IJACT*, 1(4):264–289, 2009.
14. 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 *LNCS*. Springer, 2006.
15. Hugo Krawczyk. RFC: HMAC-based Extract-and-Expand Key Derivation Function (HKDF). https://tools.ietf.org/html/rfc5869.txt, May 2010.
16. Mario Lamberger, Florian Mendel, Christian Rechberger, Vincent Rijmen, and Martin Schläffer. Rebound Distinguishers: Results on the Full Whirlpool Compression Function. In Mitsuru Matsui, editor, *ASIACRYPT*, volume 5912 of *LNCS*, pages 126–143. Springer, 2009.

17. Gaëtan Leurent, Thomas Peyrin, and Lei Wang. New Generic Attacks Against Hash-based MACs. In *ASIACRYPT*, 2013.
18. Florian Mendel, Christian Rechberger, Martin Schläffer, and Søren S. Thomsen. The Rebound Attack: Cryptanalysis of Reduced Whirlpool and Grøstl. In Orr Dunkelman, editor, *FSE*, volume 5665 of *LNCS*, pages 260–276. Springer, 2009.
19. Kazuhiko Minematsu and Yukiyasu Tsunoo. Provably Secure MACs from Differentially-Uniform Permutations and AES-Based Implementations. In Matthew J. B. Robshaw, editor, *FSE*, volume 4047 of *LNCS*, pages 226–241. Springer, 2006.
20. NESSIE. New European Schemes for Signatures, Integrity, and Encryption. IST-1999-12324. Available online at http://cryptonessie.org/.
21. Bart Preneel, René Govaerts, and Joos Vandewalle. Hash Functions Based on Block Ciphers: A Synthetic Approach. In Douglas R. Stinson, editor, *CRYPTO*, volume 773 of *LNCS*, pages 368–378. Springer, 1993.
22. Christian Rechberger and Vincent Rijmen. On Authentication with HMAC and Non-random Properties. In Sven Dietrich and Rachna Dhamija, editors, *Financial Cryptography*, volume 4886 of *LNCS*, pages 119–133. Springer, 2007.
23. Christian Rechberger and Vincent Rijmen. New Results on NMAC/HMAC when Instantiated with Popular Hash Functions. *J. UCS*, 14(3):347–376, 2008.
24. Vincent Rijmen and Paulo S. L. M. Barreto. The WHIRLPOOL Hashing Function. Submitted to NISSIE, September 2000.
25. Yu Sasaki. Meet-in-the-Middle Preimage Attacks on AES Hashing Modes and an Application to Whirlpool. In Antoine Joux, editor, *FSE*, volume 6733 of *LNCS*, pages 378–396. Springer, 2011.
26. Yu Sasaki. Cryptanalyses on a Merkle-Damgård Based MAC - Almost Universal Forgery and Distinguishing-H Attacks. In David Pointcheval and Thomas Johansson, editors, *EUROCRYPT*, volume 7237 of *LNCS*, pages 411–427. Springer, 2012.
27. Yu Sasaki, Lei Wang, Shuang Wu, and Wenling Wu. Investigating Fundamental Security Requirements on Whirlpool: Improved Preimage and Collision Attacks. In Xiaoyun Wang and Kazue Sako, editors, *ASIACRYPT*, volume 7658 of *LNCS*, pages 562–579. Springer, 2012.
28. JH. Song, R. Poovendram ad J. Lee, and T. Iwata. The AES-CMAC Algorithm, June 2006.
29. 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 *LNCS*, pages 237–253. Springer, 2008.
30. 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 *LNCS*, pages 121–133. Springer, 2009.
31. Shuang Wu, Dengguo Feng, Wenling Wu, Jian Guo, Le Dong, and Jian Zou. (Pseudo) Preimage Attack on Round-Reduced Grøstl Hash Function and Others. In Anne Canteaut, editor, *FSE*, volume 7549 of *LNCS*, pages 127–145. Springer, 2012.
32. Zheng Yuan, Wei Wang, Keting Jia, Guangwu Xu, and Xiaoyun Wang. New Birthday Attacks on Some MACs Based on Block Ciphers. In Shai Halevi, editor, *CRYPTO*, volume 5677 of *LNCS*, pages 209–230. Springer, 2009.
33. Fan Zhang and Zhijie Jerry Shi. Differential and Correlation Power Analysis Attacks on HMAC-Whirlpool. In *ITNG*, pages 359–365. IEEE Computer Society, 2011.