Improved Generic Attacks Against Hash-based MACs and HAIFA*

Itai Dinur[1] and Gaëtan Leurent[2]

[1] Département d'Informatique, École Normale Supérieure, Paris, France
Itai.Dinur@ens.fr
[2] Inria, EPI SECRET, France
Gaetan.Leurent@inria.fr

Abstract. The security of HMAC (and more general hash-based MACs) against state-recovery and universal forgery attacks was very recently shown to be suboptimal, following a series of surprising results by Leurent *et al.* and Peyrin *et al.*. These results have shown that such powerful attacks require much less than 2^{ℓ} computations, contradicting the common belief (where ℓ denotes the internal state size). In this work, we revisit and extend these results, with a focus on properties of concrete hash functions such as a limited message length, and special iteration modes. We begin by devising the first state-recovery attack on HMAC with a HAIFA hash function (using a block counter in every compression function call), with complexity $2^{4\ell/5}$. Then, we describe improved tradeoffs between the message length and the complexity of a state-recovery attack on HMAC. Consequently, we obtain improved attacks on several HMAC constructions used in practice, in which the hash functions limit the maximal message length (e.g., `SHA-1` and `SHA-2`). Finally, we present the first universal forgery attacks, which can be applied with short message queries to the `MAC` oracle. In particular, we devise the first universal forgery attacks applicable to `SHA-1` and `SHA-2`.

Keywords: Hash functions, MAC, HMAC, Merkle-Damgård, HAIFA, state-recovery attack, universal forgery attack, `GOST`, `Streebog`, SHA family.

1 Introduction

MAC algorithms are an important symmetric cryptography primitive, used to verify the integrity and authenticity of messages. First, the sender appends to the message a tag, computed from the message and a key. The receiver can recompute the tag using the key and reject the message when the computed tag does not match the received one. The main security requirement of a MAC is the resistance to existential forgery. Namely, after querying the `MAC` oracle to obtain the tags of some carefully chosen messages, it should be hard to forge a valid tag for a different message.

* Some of the work presented in this paper was done during Dagstuhl Seminar 14021.

One of the most widely used MAC algorithms in practice is HMAC, a MAC construction using a hash function designed by Bellare, Canetti and Krawczyk in 1996 [4]. The algorithm has been standardized by ANSI, IETF, ISO and NIST, and is widely deployed to secure internet communications (*e.g.* SSL, SSH, IPSec). As these protocols are widely used, the security of HMAC has been extensively studied, and several security proofs [3,4] show that it gives a secure MAC and a secure PRF up to the birthday bound (assuming good properties of the underlying compression function). At the same time, there is a simple existential forgery attack on any iterative MAC with an ℓ-bit state, with complexity $2^{\ell/2}$, matching the security proof. Nevertheless, security beyond the birthday bound for stronger attacks (such as state-recovery and universal forgery) is still an important topic.

Surprisingly, the security of HMAC beyond the birthday bound has not been thoroughly studied until 2012, when Peyrin and Sasaki described an attack on HMAC in the related-key setting [18]. Later work focused on single-key security, and included a paper by Naito, Sasaki, Wang and Yasuda [16], which described state-recovery attacks with complexity $2^{\ell}/\ell$. At Asiacrypt 2013, Leurent, Peyrin and Wang [15] gave state-recovery attacks with complexity $2^{\ell/2}$, closing the gap with the security proof. More recently, at Eurocrypt 2014, Peyrin and Wang [19] described a universal forgery attack with complexity as low as $2^{5\ell/6}$. The complexity of the universal forgery attack was further improved to $2^{3\ell/4}$ in [8], showing that even this very strong attack is possible with less than 2^{ℓ} work.

These generic attacks have also been used as a first step to build specific attacks against HMAC with the concrete hash function Whirlpool [10,9].

These very recent and surprising results show that more work is needed to better understand the exact security provided by HMAC and hash-based MACs.

1.1 Our results

In this paper, we provide several important contributions to the security analysis of HMAC and similar hash-based MAC constructions. In particular, we devise improved attacks when HMAC is used with many popular concrete hash functions, and in several cases our attacks are the first to be applicable to HMAC with the given hash function. Some results with concrete instantiations are summarized in Table 1.

As a first contribution, we focus on the HAIFA [5] mode of operation, used in many recent designs such as BLAKE [1,2], Skein [7], or Streebog [6]. The HAIFA construction uses a block counter to tweak the compression functions, such that they resemble independent random functions, in order to thwart some narrow-pipe attacks (*e.g.* the second-preimage attack of Kelsey and Schneier [13]). Indeed, the recent attacks against HMAC [15,19] use in a very strong way the assumption that the same compression function is applied to all the message blocks, and thus they cannot be applied to HAIFA. In this work, we present the first state-recovery attack on HMAC using these hash functions, whose optimal complexity is $2^{4\ell/5}$.

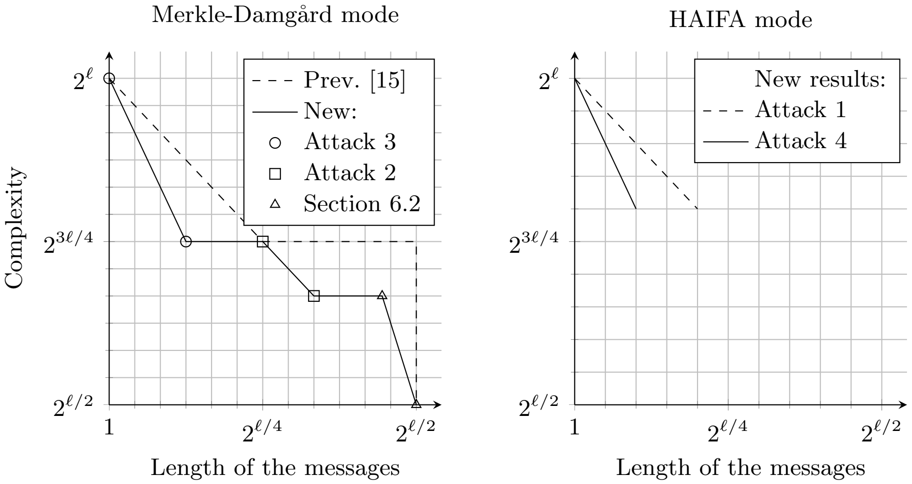


Fig. 1. Trade-offs between the message length and the complexity

In an interesting application of our state-recovery attack on HAIFA (given in Appendix A), we show how to extend it into a key-recovery attack on the new Russian standard `Streebog`, recovering the 512-bit key of `HMAC-Streebog` with a complexity of 2^{410}. This key recovery attack is similar to the one of [15] for Merkle-Damgård, and confirms its surprising observation: adding an internal checksums in a hash function (such as `Streebog`) *weakens* the design when used in HMAC, even for hash functions based on the HAIFA mode.

As a second contribution of this paper, we revisit the results of [15], and give a formal proof of the conjectures used in its short message attacks. Some of our proofs are of broad interest, as they give insight into the behavior of classical collision search algorithms for random functions. These proofs explain for the first time an interesting phenomenon experimentally observed in several previous works (such as [17]), namely, that the collisions found by such algorithms are likely to belong to a restricted set of a surprisingly small size.

Then, based on our proofs, we describe several new algorithms with various improved trade-offs between the message length and the complexity as shown in Figure 1. As many concrete hash functions restrict the message size, we obtain improved attacks in many cases: for instance, we reduce the complexity of a state-recovery attack against `HMAC-SHA-1` from 2^{120} to 2^{107} (see Table 1).

Finally, we focus on universal forgery attacks, and devise attacks using techniques which are different from those of [19]. While the attack of [19] (and its improvement in [8]) is much more efficient than exhaustive search, it requires, in an inherent way, querying the `MAC` oracle with very long messages of about $2^{\ell/2}$ blocks, and thus has limited impact in practice. On the other hand, our attacks can be efficiently applied with much shorter queries to the `MAC` oracle, and thus have many more applications. In particular, we devise the first universal forgery attack applicable to HMAC with `SHA-1` and `SHA-2` (see Table 1).

Table 1. Complexity of attacks on HMAC instantiated with some concrete hash functions. The state size is denoted as ℓ, and the maximum message length as 2^s. For the new results, we give a reference to the Attack number.

				State-recovery		Universal forgery	
Function	Mode	ℓ	s	[15]	New	[19]	New
SHA-1	MD	160	2^{55}	2^{120}	2^{107} (2)	N/A	2^{132} (6)
SHA-224	MD	256	2^{55}	2^{201}	2^{152} (3)	N/A	N/A
SHA-256	MD	256	2^{55}	2^{201}	2^{152} (3)	N/A	2^{228} (5,6)
SHA-384	MD	512	2^{118}	N/A	2^{282} (3)	N/A	N/A
SHA-512	MD	512	2^{118}	2^{394}	2^{282} (3)	N/A	2^{453} (5,6)
HAVAL	MD	256	2^{54}	2^{202}	2^{154} (3)	N/A	2^{229} (5,6)
Whirlpool	MD	512	2^{247}	2^{384}	2^{283} (7)	N/A	2^{446} (5)
BLAKE-256	HAIFA	256	2^{55}	N/A	2^{213} (4)	N/A	N/A
BLAKE-512	HAIFA	512	2^{118}	N/A	2^{419} (4)	N/A	N/A
Skein-512	HAIFA	512	2^{90}	N/A	2^{419} (4)	N/A	N/A
						Key recovery	
						[15]	New
Streebog	HAIFA+σ	512	∞	N/A	2^{419} (4)	N/A	2^{419} (8)

1.2 Framework of the attacks

In order to recover an internal state, computed by the MAC oracle during the processing of some message, we use a framework which is similar to the framework of [15]. Namely, we match states that are computed offline with (unknown) states that are computed online (during the processing of messages by the MAC oracle). However, as arbitrary states match with low probability (which does not lead to efficient attacks), we only match *special states*, which have a higher probability to be equal. These special states are the result of iterating random functions using chains, computed by applying the compression function on a fixed message from arbitrary initial states. In this paper, we exploit special states of two types, which were also exploited in [15]: states on which two evaluated chains collide, and states on which a single chain collides with itself to form a cycle. Additionally, some of our attacks (and in particular our attacks on HAIFA) use special states which are a result of the reduction of the image space that occurs when applying a fixed sequence of random functions.

As described above, after we compute special states both online and offline, we need to match them in order to recover an online state. However, since the online states are unknown, the matching cannot be performed directly, and we are forced to match the nodes indirectly using *filters*. A filter for a node (state) is a property that identifies it with high probability, i.e., once the filters of two nodes match, then the nodes themselves match with high probability. Since the complexity of the matching steps in a state-recovery attack depend on the complexity on building a filter for a node and testing a filter on a node, we are interested in building filters efficiently. In this paper, we use two types of filters: collision filters (which were also used in [15]) and diamond filters, which exploit the diamond structure (proposed in [12]) in order to build filters for a large set of nodes with reduced average complexity. Furthermore, in this paper we use a novel online construction of the diamond structure via the $\texttt{MAC}$ oracle, whereas such a structure is typically computed offline. In particular, we show that despite the fact that the online diamond filter increases the complexity of building the filter, the complexity of the actual matching phase is significantly reduced, and gives improved attacks in many cases.

Outline The paper is organized as follows: we begin with a description of HMAC in Section 2. We then describe and analyze the algorithms we use to compute special states in Section 3, and the filters we use in our attacks in Section 4. Next, we present a simple attack against HMAC with a HAIFA hash function in Section 5, and revisit the results of [15] in Section 6, presenting new trade-offs for attacks on Merkle-Damgård hash functions. In Section 7, we give more complex attacks for shorter messages. Finally, in Section 8, we present our universal forgery attacks with short queries, and conclude in Section 9.

2 HMAC and hash-based MACs

In this paper we study MAC algorithms based on a hash function, such as HMAC. HMAC is defined using a hash function H as $\texttt{HMAC}(K, M) = H(K \oplus \texttt{opad} \parallel H(K \oplus \texttt{ipad} \parallel M))$. More generally, we consider a class of designs represented by Figure 2, and defined as:

$$x_0 = I_K \qquad\qquad x_{i+1} = h_i(x_i, m_i) \qquad\qquad t = g(K, x_p, |M|).$$

The message processing updates an internal state of size ℓ, starting from a key-dependant value I_K, and the output is produced with a key-dependant finalization function g. In particular, we note that the state update does not depend on the key. Our description covers HMAC [4], Sandwich-MAC [22] and envelope-MAC [21] with any common hash function. The hash function can use the message length in the finalization process, which is a common practice, and the rounds function can depend on a block counter, as in the HAIFA mode. If the hash function uses the plain Merkle-Damgård mode, the round functions h_i are all identical (this is the model of previous attacks [15,19]).

In this work, we assume that the tag length n is larger than ℓ, so that collision in the tag result from collisions in the internal state with very high probability. This greatly simplifies the description of the attacks, and does not restrict the scope of our results. Indeed from a function $\texttt{MAC}_1(K, M)$ with an output of n bits, we can build a function $\texttt{MAC}_2(K, M)$ with a $2n$-bit output by appending message blocks [0] and [1] to M, as $\texttt{MAC}_2(K, M) = \texttt{MAC}_1(K, M \parallel [0]) \parallel \texttt{MAC}_1(K, M \parallel [1])$. Our attacks applied to $\texttt{MAC}_2$ can immediately be turned to attacks on $\texttt{MAC}_1$.

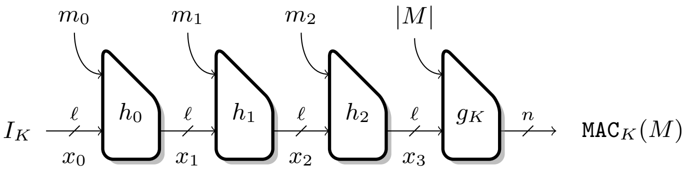


Fig. 2. Hash-based MAC with HAIFA. Only the initial value and the final transformation are keyed.

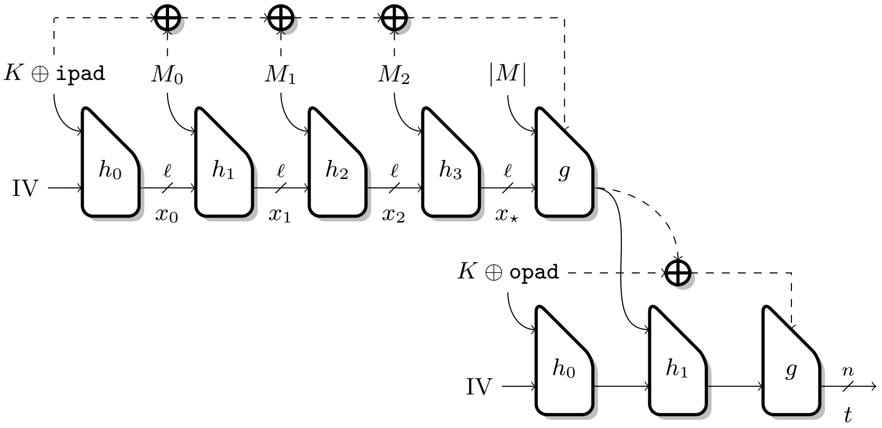


Fig. 3. `HMAC` based on a hash function with a block counter and a checksum (dashed lines).

3 Description and Analysis of Collision Search Algorithms

In this section, we describe and analyze the collision search algorithms which are used in our state-recovery attacks in order to compute special states. We then analyze these algorithms and prove the conjectures of [15]. Lemma 1 proves the first conjecture, while Lemma 3 proves the second conjecture. We also give further results in Appendix A.

3.1 Collision search algorithms

We use standard collision search algorithms, which evaluate chains starting from arbitrary points. Namely, a chain $\overrightarrow{x}$ starts from x_0, and is constructed iteratively by the equation $x_i = f_i(x_{i-1})$ up to $i = 2^s$ for a fixed value of $s \leq \ell/2$. We consider two different types of collisions between two chains $\overrightarrow{x}$ and $\overrightarrow{y}$: free-offset collisions ($x_i = y_j$ for any i, j, with all the f_i's being equal), and same-offset collisions ($x_i = y_i$).

Free-offset collision search. When searching offline for collisions in iterations of a *fixed random function* f, we evaluate 2^t chains starting from arbitrary points, and extended to length 2^s for $s \leq \ell/2$.

Assuming that $2^t \cdot 2^{t+s} \leq 2^\ell$ (i.e., $2t + s \leq \ell$), then each of the chains is not expected to collide with more than one other chain in the structure. This implies that the structure contains a total of about 2^{t+s} distinct points, and (according to the birthday paradox) we expect it to contain a total of $2^c = 2^{2(t+s)-\ell}$ collisions. We can easily recover all of these collisions in $O(2^{t+s}) = O(2^{(c+\ell)/2})$ time by storing all the evaluated points and checking for collisions in memory.

We note that we can reduce the memory requirements of the algorithm by using the parallel collision search algorithm of van Oorschot and Wiener [17]. However, in this paper, we generally focus on time complexity and do not try to optimize the memory complexity of our attacks.

Same-offset collision search. While free-offset collisions are the most general form of collisions, they cannot always be efficiently detected and exploited by our attacks. In particular, they cannot be efficiently detected in queries to the online oracle (as a collision between messages of different lengths would lead to different values after the finalization function). Furthermore, if the hash function uses the HAIFA iteration mode, it is also not clear how to exploit free-offset collisions offline, as the colliding chains do not merge after the collision (and thus we do not have any easily detectable non-random property).

In the cases above, we are forced to only use collisions that occur at the same-offset. When computing 2^t chains of length 2^s (for t not too large), a pair of chains collide at a fixed offset i with probability $2^{-\ell}$, and thus a pair of chains collide with probability $2^{s-\ell}$. As we have 2^{2t} pairs of chains, we expect to find about $2^{2t+s-\ell}$ fixed-offset collisions.

Locating collisions online. Online collisions are detected by sorting and comparing the tags obtained by querying the $\mathtt{MAC}$ oracle with chains of a fixed length 2^s. If we find two massages such that $\mathtt{MAC}(M) = \mathtt{MAC}(M')$, we can easily compute the message prefix that gives the (unknown) collision state, as described in [15]. Namely, if we denote by $M_{|i}$ the i-block prefix of M, then we find the smallest i such that $\mathtt{MAC}(M_{|i}) = \mathtt{MAC}(M'_{|i})$ using binary search. This algorithm queries the $\mathtt{MAC}$ oracle with $O(s)$ messages of length $O(2^s)$, and thus the time complexity of locating a collision online is $s \cdot 2^s = \tilde{O}(2^s)$.

3.2 Analysis of the collision search algorithms

In this section, we provide useful lemmas regarding the collision search algorithms described above. These lemmas are used in order to estimate the collision probability of special states that are calculated by our attacks and thus to bound their complexity. Lemmas 1 and 2 can generally be considered as common knowledge in the field, and their proofs are given in Appendix A. Perhaps, the most interesting results in this section are lemmas 3 and 4. These lemmas show that the probability that our collision search algorithms reach the same collision twice from different arbitrary starting points, is perhaps higher than one would expect. This phenomenon was already observed in previous works such as [17], but to the best of our knowledge, this is the first time that this lemma is formally proven. As the proof of lemma 4 is very similar to that of lemma 3, it is given in Appendix A.

Lemma 1. *Let $s \leq \ell/2$ be a non-negative integer. Let $f_1, f_2, \ldots, f_{2^s}$ be a sequence of random functions over the set of 2^ℓ elements, and $g_i \triangleq f_i \circ \ldots \circ f_2 \circ f_1$ (with the f_i being either all identical, or independently distributed). Then, the images of two arbitrary inputs to g_{2^s} collide with probability of about $2^{s-\ell}$,* i.e. $\Pr_{x,y}\left[g_{2^s}(x) = g_{2^s}(y)\right] = \Theta(2^{s-\ell})$.

Lemma 2. *Let $f_1, f_2, \ldots, f_{2^s}$ be a sequence of random functions, then the image of the function $g_{2^s} \triangleq f_{2^s} \circ \ldots \circ f_2 \circ f_1$ contains at most $\tilde{O}(2^{\ell-s})$ points.*

Lemma 3. *Let $\hat{x}$ and $\hat{y}$ be two random collisions (same-offset or free-offset) found by a collision search algorithm using chains of length 2^s, with a **fixed $\boldsymbol{\ell}$-bit function $\boldsymbol{f}$** such that $s < \ell/2$. Then $\Pr\left[\hat{x} = \hat{y}\right] = \Theta(2^{2s-\ell})$.*

Proof. First, we note that we generally have 4 cases to analyze, according to whether $\hat{x}$ and $\hat{y}$ were found using a free-offset, or a same-offset collision search algorithm. However, the number of cases can be easily reduced to 3, as we have 2 symmetric cases, where one collision is free-offset, and the other is same-offset. In this proof, we assume that $\hat{x}$ is a same-offset collision and $\hat{y}$ is a free-offset collision (this is the configuration used in our attacks). However, the proof can easily be adapted to the 2 other different settings.

We denote the starting points of the chains which collide on $\hat{x}$ by (x_0, x'_0), and the actual corresponding colliding points of the chains by (x_i, x'_i), and thus $f(x_i) = f(x'_i) = \hat{x}$. In the following, we assume that $0.25 \cdot 2^s \leq i \leq 0.75 \cdot 2^s$, which occurs with probability about 1/2 since the offset of the collision $\hat{x}$ is roughly uniformly distributed in the interval $[0, 2^s]$.[3] This can be shown using Lemma 1, as increasing the length of the chains, increases the collision probability by the same multiplicative factor.

Fixing (x_0, x'_0), we now calculate the probability that 2 chains of length 2^s, starting from arbitrary points (y_0, y'_0), collide on $\hat{x}$. This occurs if $y_0, y_1, \ldots, y_{2^s-i}$ collides with $x_0, x_1, \ldots, x_i$, and $y'_0, y'_1, \ldots, y'_{2^s-i}$ collides with $x'_0, x'_1, \ldots, x'_i$ (or vise-versa), which happens with probability $\Theta(2^{2(2s-\ell)})$ (assuming $0.25 \cdot 2^s \leq i \leq 0.75 \cdot 2^s$, all chains are of length $\Theta(2^s)$). This lower bounds the collision probability on $\hat{x}$ by $\Omega(2^{2(2s-\ell)})$. At the same time, the collision on $\hat{x}$ is also upper bounded by $O(2^{2(2s-\ell)})$, as all 4 chains are of length $O(2^s)$. We conclude that the collision probability on $\hat{x}$ is $\Theta(2^{2(2s-\ell)})$.

On the other hand, the probability that the chains starting from (y_0, y'_0) collide on any point is $\Theta(2^{2s-\ell})$. Assuming that the collision search algorithm evaluates 2^t chains such that $2t + s \leq \ell$, then each evaluated chain is not expected to collide with more than one different chain, and the pairs of chains can essentially be analyzed independently.

[3] The assumption simplifies the proof of the lower bound on the collision probability.

We denote by A the event that the chains starting from (y_0, y'_0) collide on $\hat{x}$, and by B the event that the chains starting from (y_0, y'_0) collide. We are interested in calculating the conditional probability $\Pr[A|B]$, and we have $\Pr[A|B] = \Pr[A \bigcap B]/\Pr[B] = \Pr[A]/\Pr[B] = \Theta(2^{2(2s-\ell)-(2s-\ell)}) = \Theta(2^{2s-\ell})$, as required. □

Lemma 4. *Let $\hat{x}$ and $\hat{y}$ be two arbitrary same-offset collisions found, respectively, at offsets i and j by a collision search algorithm using chains of fixed length 2^s, with* ***independent $\boldsymbol{\ell}$-bit functions $\boldsymbol{f_i}$****, such that $s < \ell/2$. Then $\Pr[(\hat{x}, i) = (\hat{y}, j)] = \Theta(2^{s-\ell})$. Furthermore, given that $i = j$, we have $\Pr[\hat{x} = \hat{y}] = \Theta(2^{2s-\ell})$.*

4 Filters

We describe the two types of filters that we use in our attacks in order to match (known) states computed offline with unknown states computed online.

4.1 Collision filters

A simple filter that we use in some of our attacks was also used in the previous work of [15]. We build a collision filter $([b], [b'])$ for a state x offline by finding message blocks $([b], [b'])$ such that the states, obtained after processing these blocks from x, collide. In order to build this filter, we find a collision in the underlying hash function by evaluating its compression function for about $2^{\ell/2}$ different messages blocks from the state x. In order to test this filter online on the unknown node x' obtained after processing a message m', we simply check whether the tags of $m' \parallel [b]$ and $m' \parallel [b']$ collide. As the tags of $m' \parallel [b]$ and $m' \parallel [b']$ collide with probability $2^{-n} < 2^{-\ell}$ if the state obtained after processing m' is not x, we can conclude that the collision filter identifies the state x with high probability.

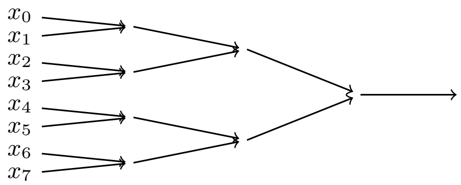
The complexity of building a collision filter offline is $O(2^{\ell/2})$. Testing the filter online requires querying the MAC oracle with $m' \parallel [b]$ and $m' \parallel [b']$, and assuming that the length of m' is $2^{s'}$, then it requires $O(2^{s'})$ time.

4.2 Diamond filters

In order to build filters for 2^t nodes, we can build a collision filter for each one of them separately, requiring a total of $O(2^{t+\ell/2})$ time. However, this process can be optimized using the diamond structure, introduced by Kelsey and Kohno in the herding attack [12]. We now recall the details of this construction.

The diamond structure is built from a set of 2^t states x_i, constructing a set of messages m_i of length $O(t)$, such that iterating the compression function from any state x_i using message m_i leads to a fixed final state y. The structure is built in $O(t)$ iterations, where each iteration processes a layer of nodes, and outputs a smaller layer to be processed by the next iteration. This process terminates once the layer contains only one node, which is denoted by y.

Starting from the first layer with 2^t points, we evaluate the compression function from each point x_i with about $2^{(\ell-t)/2}$ random message blocks. This gives a total of about $2^{(\ell+t)/2}$ random values, and we expect them to contain about 2^t collisions. Each collision allows to match two different values x_i, x_j and to send them to a common value in the next layer, such that its size is reduced to about $1/2$. The message m_i for a state x_i is constructed by concatenating the $O(t)$ message blocks on its path leading to y. According to the detailed analysis of [14], the time complexity of building the structure is is $\Theta(2^{(\ell+t)/2})$.



Once we finish building the diamond structure, we construct a standard collision filter for the final node y, using some message blocks $([b], [b'])$. Thus, building a diamond filter offline for 2^t states requires $O(2^{(\ell+t)/2})$ time, which is faster than the $O(2^{t+\ell/2})$ time required to build a collision filter for each node separately.

In order to test the filter for a state x_i (in the first layer of the diamond structure), on the unknown node x' obtained after processing a message m' online, we simply check whether the tags of $m' \parallel m_i \parallel [b]$ and $m' \parallel m_i \parallel |[b']$ collide. Assuming that the length of m' is $2^{s'}$, then the online test requires $O(t + 2^{s'})$ time.

Online diamond filter. A novel observation that we use in this paper, is that in some attacks it is more efficient to build the diamond structure online by calling the MAC oracle. Namely, we construct a diamond structure for the set of 2^t states x_i, where (the unknown) x_i is a result of querying the MAC oracle with a message M_i. Note the online construction is indeed possible, as the construction algorithm does not explicitly require the value of x_i, but rather builds the corresponding m_i by testing for collisions between the states (which can be detected according to collisions in the corresponding tags). However, testing for collisions online requires that all the messages M_i, for which we build the online diamond filter, are of the same length. Assuming that the messages M_i are of length 2^s, this construction requires $O(2^{s+(t+\ell)/2})$ calls to the compression function.

In order to test the filter for an unknown online state x_i, on a known state x', we simply evaluate the compression function from x' on $m_i \parallel [b]$ and $m_i \parallel |[b']$, and check whether the resulting two states are equal. Thus, the offline test requires $O(t)$ time.

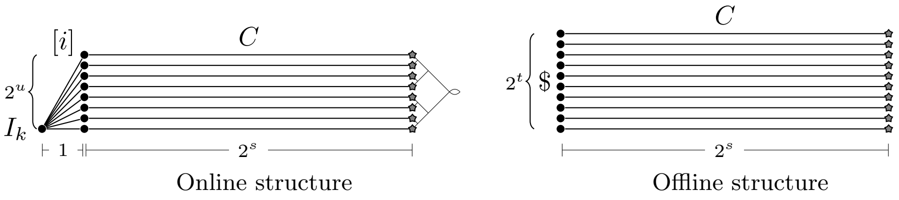
5 Internal state-recovery for NMAC and HMAC with HAIFA

In this section, we describe the first internal state-recovery attack applicable to HAIFA (which can also be used as a distinguishing-H attack). Our optimized attack has a complexity of $\tilde{O}(2^{\ell-s})$ using messages of length 2^s, but this only applies with $s \leq \ell/5$; the lowest complexity we can reach is $2^{4\ell/5}$. We note that attacks against HAIFA can also be used to attack a Merkle-Damgård hash function; this gives more freedom in the queried messages by removing the need for long series of identical blocks as in [15].

In this attack, we fix a long sequence of random functions in order to reduce the entropy of the image states, based on Lemma 1. We then use an online diamond structure to match the states computed online with states that are compute offline. The detailed attack is as follows:

Attack 1: State-recovery attack against HMAC with HAIFA
Complexity $\tilde{O}(2^{\ell-s})$, with $s \leq \ell/5$ (min: $2^{4\ell/5}$)

1. (online) Fix a message C of length 2^s. Query the oracle with 2^u messages $M_i = [i] \parallel C$. Build an online diamond filter for the set of unknown states X, obtained after M_i.
2. (offline) Starting from 2^t arbitrary starting points, iterate the compression function with the fixed message C.
3. (offline) Test each image point x', obtained in Step 2, against each of the unknown states of X. If a match is found, then with high probability the state reached after the corresponding M_i is x'.



Online structure

Offline structure

We detect a match between the grey points (✪) using the diamond test built online.

Complexity analysis. In Step 3, we match the set X of size 2^u (implicitly computed during Step 1), and a set of size 2^t (computed during Step 2). We compare 2^{t+u} pairs of points, and each pair collides with probability $2^{s-\ell}$ according to Lemma 1. Therefore, the attack is successful with high probability if $t + u \geq \ell - s$. We now assume that $t = \ell - s - u$, and evaluate the complexity of each step of the attack:

$$\textbf{Step 1:} \quad 2^{s+u/2+\ell/2} \qquad \textbf{Step 2:} \quad 2^{s+t} = 2^{\ell-u} \qquad \textbf{Step 3:} \quad 2^{t+u} \cdot u = 2^{\ell-s} \cdot u$$

The lowest complexity is reached when all the steps of the attack have the same complexity, with $s = \ell/5$. More generally, we assume that $s \leq \ell/5$ and we set $u = s$. This give an attack with complexity $\tilde{O}(2^{\ell-s})$ since $s + u/2 + \ell/2 = 3s/2 + \ell/2 \leq 4\ell/5 \leq \ell - s$.

6 New Tradeoffs for Merkle-Damgård

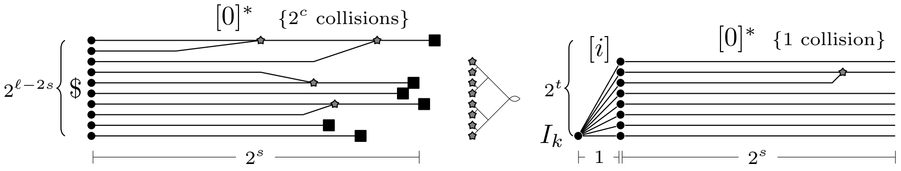
In this section, we revisit the results of [15], and give more flexible tradeoffs for various message lengths.

6.1 Trade-off based on iteration chains

In this attack, we match special states obtained using collision, based on Lemma 3. This attack extends the original tradeoff of [15] by using two improved techniques: first, while [15] used a fixed-offset offline collision search, we use a more general, free-offset offline collision search, which enables us to find collisions more efficiently. Second, while [15] used collision filters, we use a more efficient diamond filter.

Attack 2: Chain-based trade-off for HMAC with Merkle-Damgård
Complexity $O(2^{\ell-s})$, with $s \leq \ell/3$ (min: $2^{2\ell/3}$)

1. (offline) Use free-offset collision search from $2^{\ell-2s}$ starting points with chains of length 2^s, and find 2^c collisions (denoted by the set $\hat{X}$).
2. (offline) Build a diamond filter for the points in $\hat{X}$.
3. (online) Query the oracle with 2^t messages $M_i = [i] \parallel [0]^{2^s}$. Sort the tags, and locate 1 collision.
4. (online) Use a binary search to find the message prefix giving the unknown online collision state $\hat{y}$.
5. (online) Match the unknown online state $\hat{y}$ with each offline state in $\hat{X}$ using the diamond filter. If a match with $\hat{x} \in \hat{X}$ is found, then with very high probability $\hat{y} = \hat{x}$.



Offline structure

Online structure

We generate collisions offline using free-offset collision search, build a diamond filter for the collision points (⋆), and recover the state of an online collision.

Complexity analysis. In Step 1, we use free-offset collision search with $2^{\ell-2s}$ starting points and chains of length 2^s, and thus according to Section 3.1, we find $2^{\ell-2s}$ collisions (*i.e.* $c = \ell - 2s$). Furthermore, according to Lemma 3, $\hat{y} \in \hat{X}$ with high probability, in which case the attack succeeds.

In Step 3, we use fixed-offset collision search with 2^t starting points and chains of length 2^s, and thus according to Section 3.1, we find $2^{2t+s-\ell}$ collisions. As we require one collision, we have $t = (\ell - s)/2$. We now compute the complexity of each step of the attack:

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

With $s \leq \ell/3$, we have $(\ell+s)/2 \leq 2/3 \cdot \ell \leq \ell - s$, and the complexity of the attack is $O(2^{\ell-s})$.

6.2 Trade-off based on cycles

We also generalize the cycle-based state-recovery attack of [15], which uses messages of length $2^{\ell/2}$ and has a complexity of $2^{\ell/2}$. Our attack uses (potentially) shorter messages of length 2^s for $s \leq \ell/2$, and has a complexity of $2^{2\ell-3s}$. The full attack and its analysis is given in Appendix B.

7 Shorter Message Attacks

In this section, we describe more complex attacks that can reach a tradeoff of $2^{\ell-2s}$, for relatively small values of s. These attacks are useful in cases where the message length of the underlying hash function is very restricted (*e.g.* the `SHA-2` family). In order to reach a complexity of $2^{\ell-2s}$, we combine the idea of building filters in the online phase with lemmas 3 and 4.

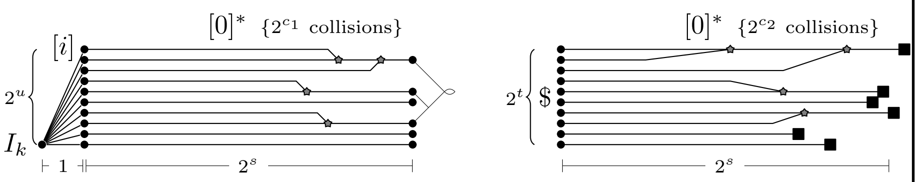
In the case of Merkle-Damgård with identical compression functions, we reach a complexity of $2^{\ell-2s}$ for $s \leq \ell/8$, *i.e.* the optimal complexity of this attack is $2^{3/4 \cdot \ell}$. With the HAIFA mode of operation, we reach a complexity of $2^{\ell-2s}$ for $s \leq \ell/10$ *i.e.* the optimal complexity of $2^{4/5 \cdot \ell}$, matching the optimal complexity of the attack of Section 5.

7.1 Merkle-Damgård

Attack 3: Short message attack for HMAC with Merkle-Damgård
Complexity $\tilde{O}(2^{\ell-2s})$, with $s \leq \ell/8$ (min: $2^{3\ell/4}$)

1. (online) Query the oracle with 2^u messages $M_i = [i] \parallel [0]^{2^s}$, and locate 2^{c_1} collisions.
2. (online) For each collision (i,j), use a binary search to find the distance (offset) μ_{ij} from the starting point to the collision, and denote the (unknown) state reach after M_i (or M_j) by y_{ij}.
   Denote the set of all y_{ij} (containing about 2^{c_1} states) by Y. Build an online diamond filter for all the states in Y.
3. (offline) Run a free-offset collision search algorithm from 2^t starting points with chains of length 2^s, and locate 2^{c_2} collisions.

4. (offline) For each offline collision $\hat{x}$, match its iterates with all points $y_{ij} \in Y$: iterate the compression function with a zero message starting from $\hat{x}$ (up to 2^s times), and match iterate $2^s - \mu_{ij}$ (i.e., $f^{2^s-\mu_{ij}}(\hat{x})$) with y_{ij} using the diamond filter. If a match is found, then with high probability $y_{ij} = f^{2^s-\mu_{ij}}(\hat{x})$.


$[0]^*$ $\{2^{c_1}$ collisions$\}$
$[i]$
2^u
I_k
1
2^s
$[0]^*$ $\{2^{c_2}$ collisions$\}$
2^t
\$
2^s

Online structure

Offline structure

We generate collisions and build a diamond filter online, and match them with collisions found offline.

Complexity analysis. Using similar analysis to Section 6.1, we have $c_1 = 2u + s - \ell$ (as a pair of chains collide at the same offset with probability $2^{s-\ell}$, and we have 2^{2u} such pairs) and $c_2 = 2t + 2s - \ell$. The attack succeeds if the sets of collisions found online and offline intersect. According to Lemma 3, this occurs with high probability if $c_1 + c_2 \geq \ell - 2s$. In the following, we assume $c_1 + c_2 = \ell - 2s$.

Step 1: $2^{u+s} = 2^{s/2+c_1/2+\ell/2}$ **Step 2:** $2^{s+c_1/2+\ell/2} = 2^{\ell-c_2/2}$

Step 3: $2^{t+s} = 2^{\ell/2+c_2/2}$ **Step 4:** $2^{c_2+s} + 2^{c_1+c_2} \cdot c_1 = 2^{c_2+s} + 2^{\ell-2s} \cdot c_1$

The best tradeoffs are achieved by balancing steps 2 and 3, *i.e.* with $c_2 = \ell/2$. This reduces the complexity to:

Step 1: $2^{3\ell/4-s/2}$ **Step 2:** $2^{3\ell/4}$

Step 3: $2^{3\ell/4}$ **Step 4:** $2^{\ell/2+s} + 2^{\ell-2s} \cdot \ell/2$

With $s \leq \ell/8$, we have $\ell/2 + s \leq 5\ell/8$ and $3\ell/4 \leq \ell - 2s$; therefore the complexity of the attack is $\tilde{O}(2^{\ell-2s})$.

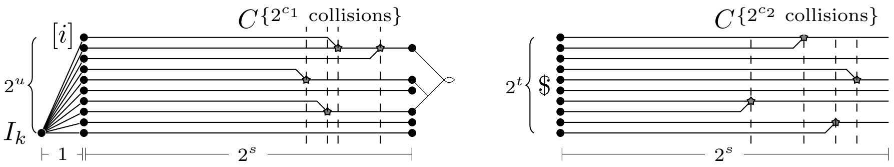
7.2 HAIFA

Since the attack is very similar to the previous attack on Merkle-Damgård, we only specify the differences between the attacks.

Attack 4: Short message attack for HMAC with HAIFA
Complexity $\tilde{O}(2^{\ell-2s})$, with $s \leq \ell/10$ (min: $2^{4\ell/5}$)

- In Step 1 of Attack 3, we fix an arbitrary suffix C of length 2^s, and use $M_i = [i] \parallel C$.

- Correspondingly, in Step 3, we use a fixed-offset collision search by iterating the compression function with C from 2^t starting points.
- In Step 4, we match each offline collision $\hat{x}$, only with online collisions that occur at the same offset as $\hat{x}$. Thus, for each $\hat{x}$, we test only the end point of its chain (at offset 2^s) with the corresponding states in Y. Note that each $\hat{x}$ is matched with $2^{c_1} \cdot 2^{-s}$ states in Y on average.



Online structure

Offline structure

We generate collisions and build an online diamond filter, and match them with offline collisions using the collision offset as a first filter.

Analysis. The attack succeeds in case there is a match between the set of collisions detected online and offline, that occurs at the same offset. According to Lemma 4, this match occurs with high probability when $c_1 + c_2 \geq \ell - s$, and thus we assume that $c_1 + c_2 = \ell - s$.

Complexity analysis. Similar to the analysis of the previous attacks, we have $c_1 = 2u + s - \ell$ and $c_2 = 2t + s - \ell$.

Step 1: $2^{u+s} = 2^{s/2+c_1/2+\ell/2}$ **Step 2:** $2^{s+c_1/2+\ell/2} = 2^{\ell-c_2/2+s/2}$

Step 3: $2^{s+t} = 2^{s/2+c_2/2+\ell/2}$ **Step 4:** $2^{c_1+c_2-s} \cdot u = 2^{\ell-2s} \cdot u$

The best tradeoffs are achieved by balancing steps 2 and 3, *i.e.* with $c_2 = \ell/2$. This reduces the complexity to:

Step 1: $2^{3\ell/4}$ **Step 2:** $2^{3\ell/4+s/2}$ **Step 3:** $2^{3\ell/4+s/2}$ **Step 4:** $2^{\ell-2s} \cdot 3\ell/4$

With $s \leq \ell/10$, we have $3\ell/4 + s/2 \leq 4\ell/5 \leq \ell - 2s$; therefore the complexity of the attack is $\tilde{O}(2^{\ell-2s})$.

8 Universal Forgery Attacks with Short Queries

We now revisit the universal forgery attack of Peyrin and Wang [19]. In this attack, the adversary receives a challenge message of length 2^t at the beginning of the game, and interacts with the oracle in order to predict the tag of the challenge. The attack of [19] has two phases, where in the first phase, the adversary recovers the internal state of the MAC at some step during the computation on the challenge. In the second phase, the adversary uses a second-preimage attack on long messages in order to generate a different message with the same tag as the challenge.

The main draw back of the attack of Peyrin and Wang (as well as its recent improvement [8]) is that its first phase uses very long queries to the `MAC` oracle, regardless of the length of the challenge. In this section, we use the tools developed in this paper to devise two universal forgery attacks which use shorter queries to the `MAC` oracle. Our first universal forgery attack has a complexity of $2^{\ell-t}$ for $t \leq \ell/7$, using queries to the `MAC` oracle of length of at most 2^{2t} (which is much smaller than $2^{\ell/2}$ for any $t \leq \ell/7$). Thus, the optimal complexity of this attack is $2^{6\ell/7}$, obtained with a challenge of length at least $2^{\ell/7}$. Our second universal forgery attack has a complexity of only $2^{\ell-t/2}$. However, it is applicable for any $t \leq 2\ell/5$, using queries to the `MAC` oracle of length of at most 2^t. Thus, this attack has an improved optimal complexity of $2^{4\ell/5}$, which is obtained with a challenge of length at least $2^{2\ell/5}$.

In order to devise our attacks, we construct different state-recovery algorithms than the one used in [19], but reuse its second phase (i.e., the second-preimage attack) in both of the attacks. Thus, in the following, we concentrate of the state-recovery algorithms, and note that since the complexity of the second phase of the attack is $2^{\ell-t}$ for any value of t, it does not add a significant factor to the time complexity.

8.1 A universal forgery attack based on the reduction of the image-set size

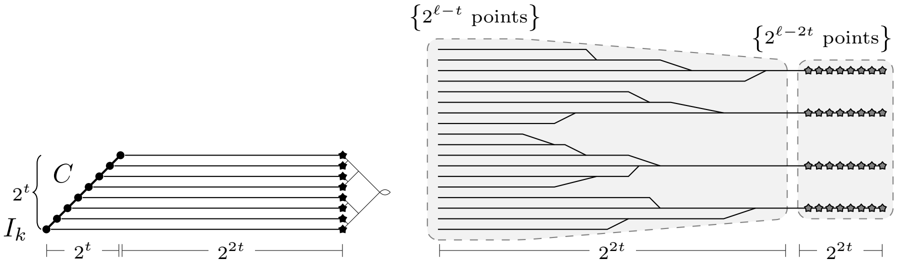
Directly matching the 2^t states of the challenge message with some states evaluated offline is too expensive. Thus, we first reduce the number of nodes we match by computing and matching the images of the states under iterations of a fixed function. After matching the images, we can efficiently match and recover on the states of the challenge message.

We denote the challenge message as C, and the first κ blocks of C as $C_{|\kappa}$. The details of the first phase of the attack are as follows.

Attack 5: Universal forgery attack based on the reduction of the image-set size (first phase)
Complexity $\tilde{O}(2^{\ell-t})$, with $t \leq \ell/7$ (min: $2^{6\ell/7}$)

1. (online) Build a collision filter for the last (unknown) state z obtained during the computation of `MAC`(C).
2. (online) Query the oracle with 2^t messages $M_i = C_{|i} \parallel [0]^{2^{2t}-i}$. Denote the set of (unknown) final states of the chains by Y. Build a diamond filter for all states in Y.
3. (offline) Compute a structure of chains containing a total of $2^{\ell-t}$ points. Each chain is extended to a maximal length of 2^{2t+1}, or until it collides with a previous chain. Consider the set X of the 2^{2t} final states of all the chains. According to Lemma 2, this set contains (no more than) about $2^{\ell-2t}$ distinct points, as all the points are in the image of $f^{2^{2t}}$.

4. (offline) Match all the points $x \in X$ with the 2^t points in Y. For each match between $x \in X$ and an online state in Y (obtained using M_i), restart the chains that merge into x, in order to locate all the points at a (backward) distance of $2^{2t} - i$ from x. Denoted this set by $\textsc{Cand}(x)$.
5. (offline) Test the candidates: for each $x' \in \textsc{Cand}(x)$, compute the state obtained by following the last $2^t - i$ blocks of the challenge message, and match this state with z using the collision filter. When a match is found, the state obtained after $C_{|i}$ is x' with high probability.



Online structure Offline structure

We efficiently detect a match between the challenge points (•) and the first part of the offline structure, by first matching X (★) and Y (✩).

Analysis. The structure of Step 3 contains $2^{\ell-t}$ points, and thus according to the birthday paradox, it covers one of the 2^t points of the challenge with high probability. In this case, the attack will successfully recover the state of the covered point with high probability, as we essentially have in the offline set X almost all $2^{\ell-2t}$ images of $f^{2^{2t}}$ (including the image of the covered point).

As X contains almost all $2^{\ell-2t}$ images of $f^{2^{2t}}$, we expect a match for every point in Y in Step 3 (a total of 2^t matches). In order to calculate the expected size of $\textsc{Cand}(x)$ in Step 5, we first calculate the expected number of the endpoints of the chains computed in Step 3 of the attack. As the chains are of length of 2^{2t+1}, we expect that after evaluating the first $2^{\ell-4t}$ chains, a constant fraction of the chains will not collide with any other chain, and thus we have (at least) about $2^{\ell-4t}$ endpoints. Since the structure contains a total of $2^{\ell-t}$ points, each endpoint is a root of a tree of average size of (at most) $2^{\ell-t-(\ell-4t)} = 2^{3t}$. This gives about $2^{3t-2t} = 2^t$ candidates nodes $\textsc{Cand}(x)$ at a fixed (backwards) distance (note that each $x' \in \textsc{Cand}(x)$ is extended with a message of length about 2^t, according to the challenge).

Complexity.

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

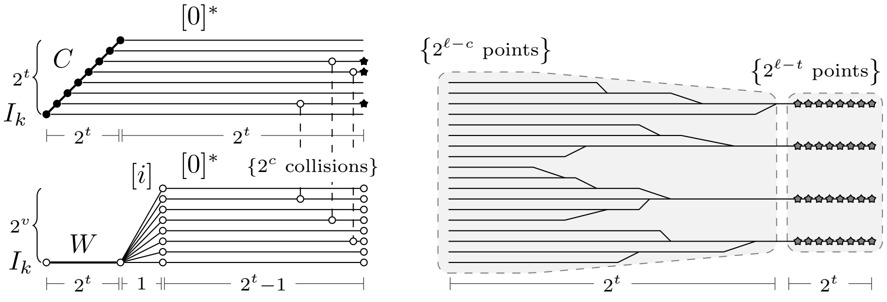
With $t \leq \ell/7$, we have $\ell/2 + 5t/2 \leq 6\ell/7 \leq \ell - t$; the complexity of the first phase of the universal forgery attack is $\tilde{O}(2^{\ell-t})$, and as the second phase has a similar complexity, this is also the complexity of the full attack.

8.2 A universal forgery attack based on collisions

In this attack, we devise a different algorithm which recovers one of the states computed during the execution of the challenge message. The main idea here is to find collisions between chains evaluated online, and directly match them with collisions obtained offline. This is different from the previous algorithm, which matched the endpoints of the chains, rather than nodes on which the collisions occur. Once the collisions are matched, similarly to the previous algorithm, we obtain a small set of candidate nodes, which we match with the actual challenge nodes.

Attack 6: Universal forgery attack based on collisions (first phase)
Complexity $O(2^{\ell-t/2})$, with $t \leq 2\ell/5$ (min: $2^{4\ell/5}$)

1. (online) Query the oracle with 2^t messages $M_i = C_{|i} \,\|\, [0]^{2^{t+1}-i}$, and sort the tags.
2. (online) Execute state-recovery Attack 2 using messages of length $min(2^t, 2^{\ell/3})$, and denote by W a message of length 2^t whose last computed state is recovered.[a]
3. (online) Query the oracle with 2^v messages $W_j = W \,\|\, [j] \,\|\, 0^{2^t-1}$, sort the tags, and locate 2^c collisions with the tags computed using the messages M_i. For each collision of tags between M_i and W_j, find the first collision point using binary search (note that the state of the collision is known, as the state obtained after processing W is known). Store all the collision states $\hat{x}_{ij}$ in a sorted list, each one next to its distance d_{ij} from $C_{|i}$.
4. (offline) Compute a structure of chains containing a total of $2^{\ell-c}$ points. Each chain is extended to a maximal length of 2^{t+1}, or until it collides with a previous chain.
5. (offline) For each offline point in the structure y which collides with an online collision $\hat{x}_{ij}$ (i.e., $y = \hat{x}_{ij}$), retrieve candidate points $\textsc{Cand}(y)$ for the state obtained after processing $C_{|i}$. This is done by computing the d_{ij}-preimage points of y in the structure (i.e., the points which are at distance d_{ij} backwards from y). Assume that for each $y = \hat{x}_{ij}$, we have an average of 2^u candidate points, and thus we have a total of at most 2^{c+u} candidate points to test in the set $\bigcup_{ij}(\textsc{Cand}(y = \hat{x}_{ij}))$. Build a diamond filter for all the 2^{c+u} candidate points.
6. (online) For each $(\hat{x}_{ij}, y)$, match the state obtained after $C_{|i}$ with all the corresponding 2^u candidate points in $\textsc{Cand}(y)$ using the diamond filter. If a match is found, then with high probability the state obtained after processing $C_{|i}$ is equal to the tested candidate.



Online structure Offline structure

We match the known points in X (★) and Y (✩) in order to detect a match between the challenge points (•) and the first part of the offline structure.

[a] In case $t > \ell/3$, we first recover the last computed state of a message of size $2^{\ell/3}$, and then complement it arbitrarily to a length of 2^t.

Analysis. In Step 3 of the attack, we find 2^c collisions between pairs of chains, where the prefix of one chain in each pair is some challenge prefix $C_{|i}$. Thus, the 2^c collisions cover 2^c such challenge prefixes, and moreover, the offline structure, computed in Step 4, contains $2^{\ell-c}$ points. Thus, according to the birthday paradox, with high probability, the offline structure covers one of the states obtained after the computation of a prefix $C_{|i}$, such that the message M_i collides with some W_j on a point $\hat{x}_{ij}$ in Step 3. Since the state obtained after the computation of $C_{|i}$ is covered by the offline structure, then $\hat{x}_{ij}$ is also covered by the offline structure, and thus the state corresponding to $C_{|i}$ will be matched as a candidate and recovered in Step 6.

In order to calculate the value of c, note that the online structure, computed in Step 1, contains 2^t chains, each of length at least 2^t, and thus another arbitrary chain of length 2^t collides with one of the chains in this structure at the same offset with probability of about $2^{2t-\ell}$. Since the structure computed in Step 3 contains 2^v such chains, the expected number of detected collisions between the structures is $2^c = 2^{2t+v-\ell}$, i.e., $c = 2t + v - \ell$.

In order to calculate the value of u, we first calculate the expected number of the endpoints of the chains computed in Step 3 of the attack. As the chains are of length of 2^{t+1}, after evaluating the first $2^{\ell-2t}$ chains, a constant fraction of the chains will not collide with any other chain, and thus we have (at least) about $2^{\ell-2t}$ endpoints. Since the structure contains a total of $2^{\ell-c}$ points, each endpoint is a root of a tree of average size of (at most) $2^{\ell-c-(\ell-2t)} = 2^{2t-c}$. This gives about $2^{2t-c-t} = 2^{t-c}$ candidates nodes at a fixed depth, i.e., $u = t - c = \ell - t - v$.

We note that the last argument we use here is heuristic, as we assume that the average number of preimages at a certain distance for the specific collision points y is similar to the average for arbitrary points. However, steps 4 and 5 are not bottlenecks of the attack (as described in the complexity analysis below), and thus even if their complexity is somewhat higher, it will not effect the complexity of the full attack. Furthermore, we can perform a more complicated matching phase, in which we iteratively build filters for the offline structure at depths about $2^{t-1}, 2^{t-2}, \ldots$, and match them with the online structure. This guarantees that the expected complexity of the attack is as claimed below.

Step 1:	2^{2t}	**Step 2:**	$max(2^{\ell-t}, 2^{2\ell/3})$
Step 3:	2^{v+t}	**Step 4:**	$2^{\ell-c} = 2^{2\ell-2t-v}$
Step 5:	$(c+u)\cdot 2^{(c+u+l)/2} = t \cdot 2^{\ell/2+t/2}$	**Step 6:**	$2^{c+u+t} = 2^{2t}$

We balance steps 3 and 4 by setting $v + t = 2\ell - 2t - v$, or $v = \ell - 3t/2$. This gives a total complexity of $O(2^{\ell-t/2})$ for any $t \leq 2\ell/5$.

9 Conclusions and Open Problems

In this paper, we provided improved analysis of HMAC and similar hash-based MAC constructions. More specifically, we devised the first state-recovery attacks on HMAC built using hash functions based on the HAIFA mode, and provided improved trade-offs between the message length and the complexity of state-recovery attacks for HMAC built using Merkle-Damgård hash functions. Finally, we presented the first universal forgery attacks which can be applied with short queries to the `MAC` oracle. Since it is widely deployed, our attacks have many applications to HMAC constructions used in practice, built using `GOST`, the SHA family, and other concrete hash functions.

Our results raise several interesting future work items such as devising efficient universal forgery attacks on HMAC built using hash functions based on the HAIFA mode, or proving that this mode provides resistance against such attacks. At the same time, there is also a wide gap between the complexity of the best known attacks and the security proofs for HMAC built using Merkle-Damgård hash functions. For example, the best universal forgery attacks on these MACs are still significantly slower than the birthday bound, which is the security guaranteed by the proofs.

References

1. Aumasson, J.P., Henzen, L., Meier, W., Phan, R.C.W.: SHA-3 proposal BLAKE. Submission to NIST (2008/2010), `http://131002.net/blake/blake.pdf`
2. Aumasson, J.P., Neves, S., Wilcox-O'Hearn, Z., Winnerlein, C.: BLAKE2: Simpler, Smaller, Fast as MD5. In: Jr., M.J.J., Locasto, M.E., Mohassel, P., Safavi-Naini, R. (eds.) ACNS. Lecture Notes in Computer Science, vol. 7954, pp. 119–135. Springer (2013)
3. Bellare, M.: New Proofs for. In: Dwork, C. (ed.) CRYPTO. Lecture Notes in Computer Science, vol. 4117, pp. 602–619. Springer (2006)
4. Bellare, M., Canetti, R., Krawczyk, H.: Keying Hash Functions for Message Authentication. In: Koblitz, N. (ed.) CRYPTO. Lecture Notes in Computer Science, vol. 1109, pp. 1–15. Springer (1996)

5. Biham, E., Dunkelman, O.: A Framework for Iterative Hash Functions - HAIFA. IACR Cryptology ePrint Archive, Report 2007/278 (2007)
6. Dolmatov, V., Degtyarev, A.: GOST R 34.11-2012: Hash Function. RFC 6986 (Informational) (Aug 2013), `http://www.ietf.org/rfc/rfc6986.txt`
7. Ferguson, N., Lucks, S., Schneier, B., Whiting, D., Bellare, M., Kohno, T., Callas, J., Walker, J.: The Skein hash function family. Submission to NIST (2008/2010), `http://skein-hash.info`
8. Guo, J., Peyrin, T., Sasaki, Y., Wang, L.: Updates on Generic Attacks against HMAC and NMAC. In: CRYPTO (2014)
9. Guo, J., Sasaki, Y., Wang, L., Wang, M., Wen, L.: Equivalent Key Recovery Attacks against HMAC and NMAC with Whirlpool Reduced to 7 Rounds. In: FSE (2014)
10. Guo, J., Sasaki, Y., Wang, L., Wu, S.: Cryptanalysis of HMAC/NMAC-Whirlpool. In: Sako and Sarkar [20], pp. 21–40
11. Joux, A.: Multicollisions in Iterated Hash Functions. Application to Cascaded Constructions. In: Franklin, M.K. (ed.) CRYPTO. Lecture Notes in Computer Science, vol. 3152, pp. 306–316. Springer (2004)
12. Kelsey, J., Kohno, T.: Herding Hash Functions and the Nostradamus Attack. In: Vaudenay, S. (ed.) EUROCRYPT. Lecture Notes in Computer Science, vol. 4004, pp. 183–200. Springer (2006)
13. Kelsey, J., Schneier, B.: Second Preimages on n-Bit Hash Functions for Much Less than 2^n Work. In: Cramer, R. (ed.) EUROCRYPT. Lecture Notes in Computer Science, vol. 3494, pp. 474–490. Springer (2005)
14. Kortelainen, T., Kortelainen, J.: On Diamond Structures and Trojan Message Attacks. In: Sako and Sarkar [20], pp. 524–539
15. Leurent, G., Peyrin, T., Wang, L.: New Generic Attacks against Hash-Based MACs. In: Sako and Sarkar [20], pp. 1–20
16. Naito, Y., Sasaki, Y., Wang, L., Yasuda, K.: Generic State-Recovery and Forgery Attacks on ChopMD-MAC and on NMAC/HMAC. In: Sakiyama, K., Terada, M. (eds.) IWSEC. Lecture Notes in Computer Science, vol. 8231, pp. 83–98. Springer (2013)
17. van Oorschot, P.C., Wiener, M.J.: Parallel Collision Search with Cryptanalytic Applications. J. Cryptology 12(1), 1–28 (1999)
18. Peyrin, T., Sasaki, Y., Wang, L.: Generic Related-Key Attacks for HMAC. In: Wang, X., Sako, K. (eds.) ASIACRYPT. Lecture Notes in Computer Science, vol. 7658, pp. 580–597. Springer (2012)
19. Peyrin, T., Wang, L.: Generic Universal Forgery Attack on Iterative Hash-based MACs. In: Eurocrypt (2014)
20. Sako, K., Sarkar, P. (eds.): Advances in Cryptology - ASIACRYPT 2013 - 19th International Conference on the Theory and Application of Cryptology and Information Security, Bengaluru, India, December 1-5, 2013, Proceedings, Part II, Lecture Notes in Computer Science, vol. 8270. Springer (2013)
21. Tsudik, G.: Message authentication with one-way hash functions. SIGCOMM Comput. Commun. Rev. 22(5), 29–38 (Oct 1992), `http://doi.acm.org/10.1145/141809.141812`
22. Yasuda, K.: "Sandwich" Is Indeed Secure: How to Authenticate a Message with Just One Hashing. In: Pieprzyk, J., Ghodosi, H., Dawson, E. (eds.) ACISP. Lecture Notes in Computer Science, vol. 4586, pp. 355–369. Springer (2007)

A Extended Analysis of Collision and Cycle Search Algorithms

In this section, we prove the lemmas of Section A, in addition to further results. The collision probabilities in the lemmas are estimated up to a constant factor (using the Θ notations), but in order to upper bound the running time of our algorithms, we only need to lower bound these collision probabilities. Thus, for the sake of simplicity, we only prove lower bounds on collision probabilities (formulated using the Θ notation) in this paper, and note that the matching upper bounds can be proven by standard probabilistic arguments.

Lemma 1 (restated). *Let $s \leq \ell/2$ be a non-negative integer. Let $f_1, f_2, \ldots, f_{2^s}$ be a sequence of random functions over the set of 2^ℓ elements, and $g_i \triangleq f_i \circ \ldots \circ f_2 \circ f_1$ (with the f_i being either all identical, or independently distributed). Then, the images of two arbitrary inputs to g_{2^s} collide with probability of about $2^{s-\ell}$,* i.e. $\Pr_{x,y}\left[g_{2^s}(x) = g_{2^s}(y)\right] = \Theta(2^{s-\ell})$.

Proof. Let x and y be two arbitrary points, $x_i = g_i(x)$ and $y_i = g_i(y)$ (or equivalently $x_0 = x, x_i = f_i(x_{i-1})$ and $y_0 = y, y_i = f_i(y_{i-1})$). As f_i is a random function, at each step of the iteration, there is a probability of $2^{-\ell}$ that the two chains collide, given that they have not collides before (otherwise, they collide with probability 1). This proves the lower bound $\Pr[g_{2^s}(x) = g_{2^s}(y)] \leq 1 - (1 - 2^{-\ell})^{2^s} = \Omega(2^{s-\ell})$ □

Lemma 2 (restated). *Let $f_1, f_2, \ldots, f_{2^s}$ be a sequence of random functions, then the image of the function $g_{2^s} \triangleq f_{2^s} \circ \ldots \circ f_2 \circ f_1$ contains at most $\tilde{O}(2^{\ell-s})$ points.*

Proof. Let $\alpha \geq 1$ be a parameter, and consider $\alpha \cdot (2^{\ell-s})$ inputs to g_{2^s} (denoted by x_j for $1 \leq j \leq \alpha \cdot (2^{\ell-s})$) that generate distinct images. Let x be a different input to g_{2^s}, then for each j, $\Pr(g_{2^s}(x) = g_{2^s}(x_j)) \approx 2^{s-\ell}$. Thus, for $\alpha = 1$, $g_{2^s}(x)$ collides with one of the $(2^{\ell-s}$ images with constant probability, and for a general $\alpha > 1$, this collision probability is about $1 - e^{-\alpha}$. For $\alpha = \ell$, the probability that $g_{2^s}(x)$ is a new distinct image become exponentially small in ℓ, and this allows us to take a union bound on all the inputs x (whose number is about 2^ℓ), and prove the lemma.

Lemma 4 (restated). *Let $\hat{x}$ and $\hat{y}$ be two arbitrary same-offset collisions found, respectively, at offsets i and j by a collision search algorithm using chains of fixed length 2^s, with* **independent $\boldsymbol{\ell}$-bit functions $\boldsymbol{f_i}$**, *such that $s < \ell/2$. Then* $\Pr\left[(\hat{x}, i) = (\hat{y}, j)\right] = \Theta(2^{s-\ell})$. *Furthermore, assuming that $i = j$, we have* $\Pr\left[\hat{x} = \hat{y}\right] = \Theta(2^{2s-\ell})$.

Proof. The proof follows essentially the same line of arguments as the proof of Lemma 3. We assume that $0.25 \cdot 2^s \leq i \leq 0.75 \cdot 2^s$ (which occurs with probability 1/2), and fix the collision $\hat{x}$. We denote by A the event that chains starting from arbitrary points (y_0, y'_0) collide on $\hat{x}$ at offset i, and by B the event that the chains starting from (y_0, y'_0) collide at an arbitrary offset j.

We have $\Pr[B] = \Theta(2^{s-\ell})$ (see Lemma 1) and $\Pr[A] = \Omega(2^{2(s-\ell)})$, and thus $\Pr[A|B] = \Pr[A \bigcap B]/\Pr[B] = \Pr[A]/\Pr[B] = \Omega(2^{s-\ell})$, which is the claimed lower bound.

When assuming that $i = j$, we need to change the definition of event B such that the chains starting from (y_0, y'_0) collide at the fixed offset i. This gives $\Pr[B] = \Theta(2^{-\ell})$ and $\Pr[A|B] = \Pr[A \bigcap B]/\Pr[B] = \Pr[A]/\Pr[B] = \Omega(2^{2s-\ell})$, which is the claimed lower bound. □

Definition 1. *A u-deep collision is a node with two distinct preimages, both of them being u-th images.*

Lemma 5. *A random mapping f has at most $\tilde{O}(2^{\ell}/u^2)$ u-deep collisions.*

Proof. Consider a structure of $\alpha \cdot 2^{\ell}/u^2$ chains, starting from $\alpha \cdot 2^{\ell}/u^2$ terminal nodes (i.e., nodes with no preimages) for $\alpha \geq 1$, and generating $\alpha \cdot 2^{\ell}/u^2$ u-deep collisions. This structure contains at least $\alpha \cdot 2^{\ell}/u$ distinct points. Consider another chain, starting from a different terminal node that is not contained in the initial structure. This chain will collide with the structure of $\alpha \cdot 2^{\ell}/u$ points within the first $u-1$ iterations, with probability of about $1 - e^{-\alpha}$, and in this case, it will not generate an additional u-deep collision. We choose $\alpha = \ell$ and consider all the chains starting from all the terminal nodes which are not contained in the initial structure. Taking a union bound on all these chains, we conclude that with high probability, none of the chains will generate an additional u-deep collision, implying that f has at most $\ell \cdot 2^{\ell}/u^2$ u-deep collisions.

Cycle Search. Cycles are created when a chain collides with itself while iterating a fixed function f. In order to search offline for a cycle of length $O(2^s)$ (for $s \leq \ell/2$), we evaluate $2^{\ell-2s}$ chains starting from arbitrary points, and extended to length 2^s. The probability that a chain collides with itself to form a cycle is equal (up to a constant factor) to the probability that its first half (of length 2^{s-1}) collides with its second half, which occurs with probability $\Theta(2^{2s-\ell})$. Thus, we expect to find a cycle within the evaluated $2^{\ell-2s}$ chains.

Lemma 6. *Let $\hat{x}$ be the entry point of an arbitrary cycle found by the cycle search algorithm for the fixed ℓ-bit function f, using chains of fixed length 2^s such that $s \leq \ell/2$. Let y_0 be an arbitrary point, and define the chain $y_{i+1} = f(y_i)$ for $i \in \{0, 1, \ldots, 2^s - 1\}$. Then $\Pr[\exists i | \hat{x} = y_i] = \Theta(2^{2s-\ell})$.*

Proof. We denote the starting points of the chain which collides (cycles) on $\hat{x}$ by x_0, and the actual corresponding colliding points of the chain by (x_i, x_j) (assuming $i < j$), and thus $f(x_i) = f(x_j) = \hat{x}$. In the following, we assume that $0.25 \cdot 2^s \leq i \leq 0.75 \cdot 2^s$, which occurs with constant probability. In order for the event $\exists i | \hat{x} = y_i$ to occur, it is sufficient that $y_0, y_1, \ldots, y_{2^s - i}$ collides with $x_0, x_1, \ldots, x_i$. This occurs with probability $\Omega(2^{2s-\ell})$ (assuming $0.25 \cdot 2^s \leq i \leq 0.75 \cdot 2^s$, the two chains are of length $\Theta(2^s)$). □

B State-Recovery Based on Cycles

The original cycle-based state-recovery attack of [15] exploits the main cycle of approximate length $2^{\ell/2}$ in the graph of the random mapping, in order to construct two colliding messages of the same length (thus having equal tags, which can be detected at the output). Our attack uses the same idea with shorted messages, and we refer the reader to [15] for a detailed description of the original attack.

Attack 7: Cycle-based trade-off for HMAC with Merkle-Damgård
Complexity $O(2^{2l-3s})$, with $s \leq \ell/2$ (min: $2^{\ell/2}$)

1. (offline) Search for a cycle in the functional graph of $h_{[0]}$, using the algorithm of Section 3.1 with chains of length 2^s. Denote the length of the cycle by L, and its entry point by $\hat{x}$.
2. (online) For different values of the message block $[b]$, query the `MAC` oracle with two messages $M = [b] \parallel [0]^{2^s} \parallel [1] \parallel [0]^{2^s+L}$ and $M' = [b] \parallel [0]^{2^s+L} \parallel [1] \parallel [0]^{2^s}$ (both of length $1+2^s+1+2^s+L = 2+2^{s+1}+L$), until $\texttt{MAC}(M) = \texttt{MAC}(M')$.
3. (online) Find the first point on which M and M' collide using binary search (i.e., detect the online collision, as described in Section 3.1). With high probability, the online collision state is equal to $\hat{x}$.

Complexity analysis. First, both M, M' are of the same length $2+2^{s+1}+L < 2^{s+2}$, and thus we can detect a collision in their final states by comparing the output tags in Step 2. In order for M, M' to collide on the final state, it is sufficient that two conditions occur simultaneously: first, the states obtained after evaluating the prefixes $[b] \parallel [0]^{2^s}$ and $[b] \parallel [0]^{2^s+L}$, collide. This occurs if one of the states computed during the evaluation of $[b] \parallel [0]^{2^s}$ collides with $\hat{x}$ (and thus enters the cycle of length L), which has probability $\theta(2^{2s-\ell})$ according to Lemma 6. Second, the states obtained after evaluating the suffixes $[1] \parallel [0]^{2^s+L}$ and $[1] \parallel [0]^{2^s}$, collide. Assuming that the states obtained after evaluating the prefixes collide, similarly to the previous case, this occurs if one of the states computed during the evaluation of $[1] \parallel [0]^{2^s}$ collides with $\hat{x}$. Again, this event occurs with probability $\theta(2^{2s-\ell})$ according to Lemma 6.

Thus, the success probability of Step 2 is $\Omega(2^{2(2s-\ell)})$, and we need to repeat it for $O(2^{2(\ell-2s)})$ different values of $[b]$ for the attack to succeed with high probability. Consequently, the time complexity of Step 2 is $O(2^{2(\ell-2s)+s}) = O(2^{2\ell-3s})$. The time complexity of all the steps is summarized below.

Step 1: $2^{\ell-2s+s} = 2^{\ell-s}$ **Step 2:** $2^{2\cdot(\ell-2s)+s} = 2^{2\ell-3s}$ **Step 3:** $2^s \cdot 2^{\ell-2s} \cdot s = 2^{\ell-s} \cdot s$

Since $s \leq \ell/2$, the complexity of the attack is $O(2^{2\ell-3s})$.

C Key recovery attack on HMAC with `GOST` R 34.11-2012

In [15], the state-recovery attack on HMAC with a Merkle-Damgård hash function (with complexity $2^{\ell/2}$) is extended into a key-recovery attack, in case the hash function uses an internal checksum like the `GOST` R 34.11-94 hash function (with complexity $2^{3/4 \cdot \ell}$). In this section, we show that a similar attack can be applied to a hash function based on HAIFA with an internal checksum. Namely, the state-recovery attack (with complexity $2^{4/5 \cdot \ell}$) can be extended into a key-recovery attack (with complexity $2^{4/5 \cdot \ell}$).

In particular, this attack is applicable to the recent proposal `GOST` R 34.11-2012, and gives a key-recovery attack with complexity 2^{410} for the 512-bit version. This result shows that HMAC-`GOST` R 34.11-2012 is significantly weaker than HMAC-SHA-3-512.

C.1 Description of the attack

The attack uses the same framework as [15], exploiting the structure of hash functions with a checksum. We target the finalization function in the first hash function call: the state value can be recovered using the previous state-recovery attacks, and exploiting the fact that the checksum value is key dependant, but can be controlled by injecting differences in the message: $\sigma = K \oplus \texttt{opad} \oplus \texttt{Sum}^{\oplus}(M)$. This allows for attacks which are somewhat similar to related-key attacks.

More precisely, we first generate a large set of messages of length L, leading to the same state $x_\star$, but with different checksums σ. Then, we consider collisions in the function $\sigma \mapsto g(x_\star, L, \sigma)$, which can be detected using online calls to the MAC oracle. At the same time, we can also generate collisions offline since $x_\star$ and L are known. Moreover, if we locate two collisions with the same difference in the σ input, then there is a high probability that the actual input pairs are the same (on average, we expect a single collision with a fixed difference). If we find an online collision and an offline collision with the same difference, we can therefore recover the value of K using $\sigma = K \oplus \texttt{opad} \oplus \texttt{Sum}^{\oplus}(M)$.

C.2 Detailed attack process

The first step of the attack is to use the state-recovery attack of Section 7.2. In order to deal with the checksum, we modify the attack so that we only look for collisions between pairs of messages with same checksum:

- In step 1, we use $M_i = [i] \parallel [i] \parallel C$
- In step 2, when building the diamond structure, we extend the messages by $m \parallel m$. We do the same when building a collision pair for the end point of the diamond.
- For the offline steps, we ignore the checksum, and only look for collisions in the iterated state.

Attack 8: Key recovery attack against HMAC with a GOST-like hash function
Complexity $\tilde{O}(2^{4\ell/5})$

0. Use the Attack 4 to recover the state x_1 after some message M_1, with $|M_1| = \ell/10$.
1. (offline) Starting from state x_1, use Joux's multicollision attack [11] to generate a set of $2^{7\ell/10}$ messages that all collide on the internal state before the checksum block, but with different checksums. Denote the final state as $x_\star$, and the length of the messages as L.
2. (online) Query the set of messages from Step 1 to HMAC and collect collisions ($2^{2\ell/5}$ collisions are expected). For each collision (M, M'), compute the checksum difference $\Delta M = \mathtt{Sum}^{\oplus}(M) \oplus \mathtt{Sum}^{\oplus}(M')$, and store $(\Delta M, \mathtt{Sum}^{\oplus}(M))$.
3. (offline) Choose a set of $2^{4\ell/5}$ one-block random checksums σ, compute $g(x_\star, L, \sigma)$ and collect collisions ($2^{3\ell/5}$ collisions are expected). For each collision (σ, σ'), compute the difference $\sigma \oplus \sigma'$ and compare it with the stored ΔM from Step 3. If a match is found, consider $\mathtt{Sum}^{\oplus}(M) \oplus \sigma$ and $\mathtt{Sum}^{\oplus}(M) \oplus \sigma'$ as potential key candidates, and verify them using a known tag.

Analysis. Since we have $2^{2\ell/5}$ collisions in Step 2 and collisions $2^{3\ell/5}$ in step 3, there is a high probability to find a match and recover the key.

Complexity. The total complexity is $\tilde{O}(2^{4\ell/5})$:

Step 0:	$\tilde{O}(2^{4\ell/5})$	**Step 1:**	$\ell \cdot 2^{\ell/2}$
Step 2:	$2^{\ell/10+7\ell/10} = 2^{4\ell/5}$	**Step 3:**	$2^{4\ell/5}$

Comparison with previous works. In [15], the attack uses a specific property of the finalization of GOST R 34.11-94: the message length is only used as a padding block, processed with the message input of the compression function. As a result, it is possible to build messages including a padding block, and to deduce the state of a short message from the state of a long message.

In general, the message length is used through a different function than the message blocks (this is the case in GOST R 34.11-2012), and we cannot recover the state of a short message as easily. In particular, the complexity of the state-recovery attack for short messages is an important factor: an attack with complexity $2^{\ell-s}$ as in Section 5 (or in [15], for the Merkle-Damgård construction) can only reach $2^{5\ell/6}$ for the key recovery, while the attacks with complexity $2^{\ell-2s}$ (described in Section 7) allow to reach a complexity of $2^{4\ell/5}$.