

Improved Generic Attacks Against Hash-Based MACs and HAIFA

Itai Dinur, Gaëtan Leurent

▶ **To cite this version:**

Itai Dinur, Gaëtan Leurent. Improved Generic Attacks Against Hash-Based MACs and HAIFA. Algorithmica, 2017, Special Issue: Algorithmic Tools in Cryptography, 79 (4), pp.1161–1195. 10.1007/s00453-016-0236-6 . hal-01407953

HAL Id: hal-01407953
https://inria.hal.science/hal-01407953

Submitted on 2 Dec 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Algorithmica
DOI 10.1007/s00453-016-0236-6

Improved Generic Attacks Against Hash-based MACs and HAIFA

Itai Dinur · Gaëtan Leurent

Received: 1 July 2015 / Accepted: 21 October 2016
This is the author version submitted by the authors .
The published version is available at DOI 10.1007/s00453-016-0236-6.

Abstract The security of `HMAC` (and more general hash-based MACs) against state-recovery and universal forgery attacks was shown to be suboptimal, following a series of results by Leurent *et al.* and Peyrin *et al.*. These results have shown that such powerful attacks require significantly 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 concrete hash functions that limit the message length, and apply 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` with a Merkle-Damgård hash function. Consequently, we obtain improved attacks on several `HMAC` constructions used in practice, in which the hash functions limits 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`.

Despite their theoretical interest, our attacks do not seem to threaten the practical security of the analyzed concrete `HMAC` constructions.

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

This paper in an extended version of [6], presented at CRYPTO 2014.

Itai Dinur
Department of Computer Science, Ben-Gurion University, Beer-Sheva, Israel
E-mail: dinuri@cs.bgu.ac.il

Gaëtan Leurent
Inria, EPI SECRET, France
E-mail: Gaetan.Leurent@inria.fr

1 Introduction

MAC algorithms are an important symmetric cryptography primitive, used to verify the integrity and authenticity of messages. The sender of a message uses a MAC function to compute a tag from the message and a shared secret key. The tag is appended to the message and the receiver can recompute the tag using the key, and reject the message when it 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 for an adversary to forge a valid tag for a different message.

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/TLS, 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 [23]. Later work focused on single-key security, and included a paper by Naito, Sasaki, Wang and Yasuda [21], which described state-recovery attacks with complexity $2^{\ell}/\ell$. At Asiacrypt 2013, Leurent, Peyrin and Wang [19] gave state-recovery attacks with complexity $2^{\ell/2}$, closing the gap with the security proof. Later, at Eurocrypt 2014, Peyrin and Wang [24] described a universal forgery attack with complexity as low as $2^{5\ell/6}$, a result that was further improved to $2^{3\ell/4}$ at CRYPTO 2014 [11], showing that even this very strong attack is possible with significantly less than 2^{ℓ} work.

Some of the generic attacks have also been used as a first step to build specific attacks against `HMAC` with the concrete hash function Whirlpool [12, 13].

These results show that more work is needed to better understand the exact security provided by `HMAC` and hash-based MACs in general.

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`

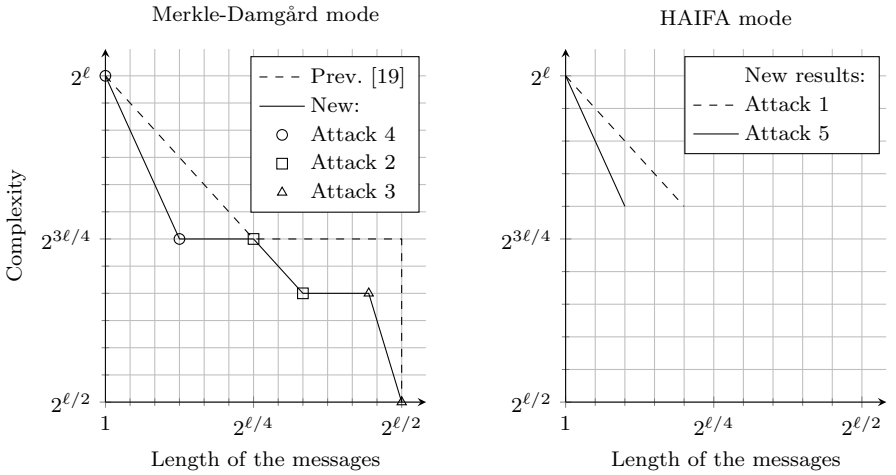


Fig. 1 Tradeoffs between the message length and the complexity for state-recovery attacks

with the given hash function. Some results with concrete instantiations are summarized in Table 1.[1]

As a first contribution, we focus on the HAIFA [5] mode of operation, used in many hash function designs such as BLAKE [1,2], Skein [8], or Streebog [7]. 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 [16]). Indeed, the previous attacks against `HMAC` [19,24] 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 $2^{4\ell/5}$.

In an interesting application of our state-recovery attack on HAIFA (given in Section 8), 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^{417}. This key-recovery attack is similar to the one of [19] for Merkle-Damgård, and confirms its surprising observation: adding 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 the full version of [19] for Merkle-Damgård hash functions (given in [18]), and we prove the conjectures used in its short message attacks. Some of our proofs are of independent 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

[1] We elaborate on our complexity evaluation in the next Subsection.

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. The figures omit small constants (following the analysis in this paper and previous works), but take into account polynomial factors in ℓ.

				State-recovery		Universal forgery	
Function	Mode	ℓ	s	[19]	New	[24]	New
SHA-1	MD	160	2^{55}	2^{120}	2^{107} (2)	N/A	2^{132} (8)
SHA-224	MD	256	2^{55}	2^{201}	2^{192} (4)	N/A	N/A
SHA-256	MD	256	2^{55}	2^{201}	2^{192} (4)	N/A	2^{228} (7,8)
SHA-512	MD	512	2^{118}	2^{394}	2^{384} (4)	N/A	2^{453} (7,8)
HAVAL	MD	256	2^{54}	2^{202}	2^{192} (4)	N/A	2^{229} (7,8)
WHIRLPOOL	MD	512	2^{247}	2^{384}	2^{283} (3)	N/A	2^{419} (8)
BLAKE-256	HAIFA	256	2^{55}	N/A	2^{213} (5)	N/A	N/A
BLAKE-512	HAIFA	512	2^{118}	N/A	2^{419} (5)	N/A	N/A
Skein-512	HAIFA	512	2^{90}	N/A	2^{419} (5)	N/A	N/A
						Key-recovery	
						[19]	New
Streebog	HAIFA+σ	512	∞	N/A	2^{417} (5,B)	N/A	2^{417} (6,B)

(such as [22]), 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 tradeoffs 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} and HMAC-WHIRLPOOL from 2^{384} to 2^{283} (see Table 1).

Finally, we focus on universal forgery attacks, where the previous attacks of [11,24] are much more efficient than exhaustive search, but require in an inherent way to query the MAC oracle with very long messages of about $2^{\ell/2}$ blocks. Thus, these attacks cannot be applied to many concrete hash functions that limit the message size. On the other hand, our techniques give rise to attacks that can be efficiently applied with much shorter queries to the MAC oracle, and therefore are more widely applicable to concrete hash functions. In particular, we devise the first universal forgery attack applicable to HMAC with SHA-1 and SHA-2 (see Table 1).

1.2 Framework of the attacks

In order to recover an internal state computed by the MAC oracle during the processing of some message (namely, mount a state-recovery attack), we use a framework which is similar to the one of [19]. 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 two types of special states which were also exploited in [19]: states on which two evaluated chains collide, and states on which a single chain collides with itself to form a cycle. We also introduce a third type of special states, which result from the reduction of the image space that occurs when applying a fixed sequence of random functions. This is used in some of our new attacks, and in particular against HAIFA.

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 depends on the complexity of 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 [19]) and diamond filters, which exploit the diamond structure (introduced in [15]) in order to build filters for a large set of nodes with reduced average complexity. In fact, we use a novel online construction of the diamond structure via the 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.

1.2.1 Complexity evaluation

The complexity of our attacks is calculated in terms of the number of compression function evaluations of the underlying hash functions. Similarly to related papers in the field, we assume that sorting a table can be performed in linear time, while searching a sorted table takes constant time. As the complexities of all our attacks are exponential in the state size ℓ, we mostly use the big-O and soft-O ($\tilde{O}$) notation to estimate them (which is common practice in analysis of exponential-time algorithms). These estimations ignore small constants and polynomial factors in ℓ (which are generally linear in our attacks). An exception to this is the more precise complexity evaluation for the attack on `HMAC-Streebog`, given in Appendix B.

1.2.2 Outline

The paper is organized as follows. We begin with a description of `HMAC` in Section 2. We then describe and analyze the algorithms used 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 [19] in Section 6, presenting new tradeoffs for attacks on Merkle-Damgård hash functions. In Section 7, we give more complex attacks for shorter messages. Our key-recovery attack on HMAC with GOST R 34.11-2012 (Streebog) is described is Section 8. Finally, in Section 9, we present our universal forgery attacks with short queries, and conclude in Section 10.

2 HMAC and Hash-based MACs

In this paper we study MAC algorithms based on a hash function, such as HMAC. Using a hash function H, HMAC is defined as $\texttt{HMAC}(K, M) = H(K \oplus \texttt{opad} \,\|\, H(K \oplus \texttt{ipad} \,\|\, M))$, as shown in Figure 2. More generally, we consider a class of iterative designs based on a family of compression functions h_i and a finalization function g, represented by Figure 3. We denote the ℓ-bit internal state as x_i and the n-bit output tag as t. The message M is divided into p blocks m_i, and the MAC is computed as:

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

The message processing updates the internal state 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 [27] and envelope-MAC [26] with any common hash function. The hash function can use the message length in the finalization process, which is a common practice, and the round 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 analyzed in previous attacks [19, 24]).

In this work, we assume that with very high probability, an arbitrary collision on the tag of two messages of the same length is a result of a collision on the final internal states x_p. This greatly simplifies the description of the attacks, and does not restrict the scope of our results. This assumption can be realized by altering the original scheme such that the new tag length is made larger than ℓ. Indeed, from a function $\mathrm{MAC}_1(K, M)$ with an output of n bits, we can build a function $\mathrm{MAC}_2(K, M)$ with a $2n$-bit output by appending message blocks [0] and [1] to M, as $\mathrm{MAC}_2(K, M) = \mathrm{MAC}_1(K, M \,\|\, [0]) \,\|\, \mathrm{MAC}_1(K, M \,\|\, [1])$. Our attacks applied to MAC_2 can immediately be turned to attacks on MAC_1 with a multiplicative penalty of 2.

In our attacks, we evaluate chains of the compression function h (or h_i for HAIFA) with a fixed message input block $[b]$ (usually $[b] = [0]$), and typically simplify our notation and define $f(x) = h(x, [b])$ (or $f_i(x) = h_i(x, [b])$ for HAIFA). We assume that the function f (or each f_i) is chosen from all ℓ-bit mappings uniformly at random. This implies that our analysis captures most (but not all) choices of underlying hash functions.

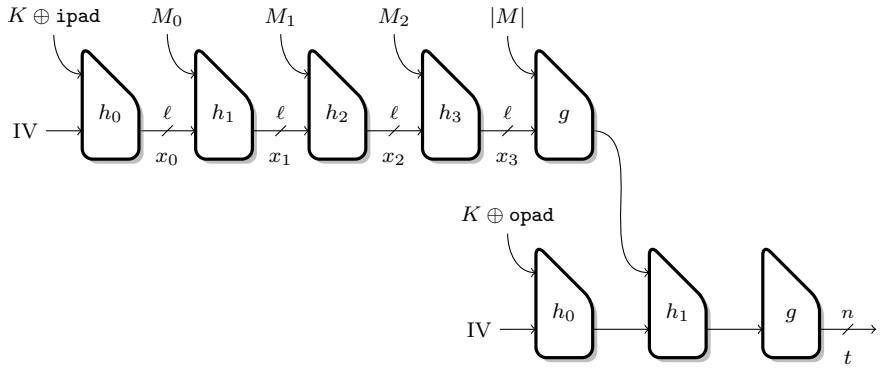


Fig. 2 `HMAC` with a HAIFA hash function. There are two hash function calls, each of them using the key at the beginning.

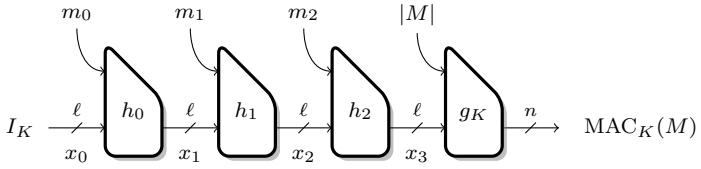


Fig. 3 Hash-based MAC with HAIFA. Only the initial value and the final transformation are keyed.

3 Description and Analysis of Collision Search Algorithms

In this section, we describe the collision search algorithms which are used in our state-recovery attacks in order to compute special states. We then analyze these algorithms (assuming they are applied to random functions) and prove the two conjectures of [19]. Lemma 1 proves the first conjecture, while Lemma 3 proves the second conjecture.

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. 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 ($x_i = y_i$).

3.1.1 Free-offset collision search

When searching offline for collisions in iterations of a *single random function* f, we evaluate 2^t chains starting from arbitrary points, and extended to length 2^s.

Assuming that $2^s \cdot 2^{t+s} \leq 2^\ell$ (*i.e.*, $2s+t \leq \ell$), then we are at (or below) the birthday paradox threshold and therefore 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 $2^c = \Theta(2^{2(t+s)-\ell})$ 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 [22]. However, in this paper, we generally focus on time complexity and do not try to optimize the memory complexity of our attacks.

3.1.2 Cycle search

Cycles are created when a chain collides with itself while iterating a single random 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.

3.1.3 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, a pair of chains collides at the same offset with probability of roughly $2^{s-\ell}$. As we have 2^{2t} pairs[2] of chains, we expect to find about $2^{2t+s-\ell}$ same-offset collisions.

We note that the computation above assumes that pairs of chains behave (almost) independently, which is the case when the total number of collisions

[2] More precisely, there are 2^{2t-1} unordered pairs of chains. However, our analysis is correct up to small constants.

in the structure of chains is relatively small. More precisely, the computation assumes that a constant fraction of the chains do not collide with any other chain, so that the chain structure contains $\Theta(2^{s+t})$ distinct points. When the iteration functions f_i are all equal, the condition reduces to $2s + t \leq \ell$ (as calculated for free-offset collision search). However, when the iteration functions f_i are distinct, we only have to consider collisions at the same offset. In order to collect the $2^{2t+s-\ell}$ same-offset collisions, we require that a constant fraction of the chains do not collide with any other chains, implying that $2t + s - \ell < t$, or $s + t \leq \ell$.

Locating collisions online. Online collisions are detected by sorting and comparing the tags obtained by querying the MAC oracle with chains of a fixed length 2^s. If we find two massages such that $\mathrm{MAC}(M) = \mathrm{MAC}(M')$, we can easily compute the message prefix that gives the (unknown) collision state, as described in [19]. Namely, if we denote by $M_{|i}$ the i-block prefix of M, then we find the smallest i such that $\mathrm{MAC}(M_{|i}) = \mathrm{MAC}(M'_{|i})$ using binary search. This algorithm queries the 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, 2 and 5 can generally be considered as common knowledge in the field. Perhaps, the most interesting results in this section are lemmas 3 and 4. These lemmas show that the probability of our collision search algorithms to 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 [22], but to the best of our knowledge, this is the first time that this phenomenon is formally proved.

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 completely independent). 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 $y_0 = y,\ y_i = f_i(y_{i-1})$). We first deal with the case of independent functions f_i. As all f_i are random functions, we have a pair of random points (x_i, y_i) for each offset. The probability that none of the pairs collide is $(1 - 2^{-\ell})^{2^s}$. The event $g_{2^s}(x) = g_{2^s}(y)$ is equivalent to having (at least) one collision between the pairs, which occurs with probability $1 - (1 - 2^{-\ell})^{2^s} = \Theta(2^{s-\ell})$ (given $s \leq \ell$).

When the functions f_i are identical, the analysis is similar, but one also needs to consider dependencies between the sequences of points which are caused by collisions $x_i = y_j$ for $i \neq j$ (and also cycles of the form $x_i = x_j$). However, since $s \leq \ell/2$, we are below (or at) the birthday bound and therefore such collisions occur with at most a (small) constant probability, and their asymptotic contribution to the value of $\Pr_{x,y}[g_{2^s}(x) = g_{2^s}(y)]$ can be neglected. □

Lemma 2 *Let $s \leq \ell/2$ be a non-negative integer. Let f be a random function, then with high probability the image size of the function f^i is $O(2^\ell/i)$.*

Proof We will show that the expected image size of f^i is upper bounded by $2 \cdot 2^\ell/i$. Since the image size is positive, this upper bound proves the lemma, as for some $\alpha > 1$ the probability that the image size is more than $2\alpha \cdot 2^\ell/i$ is at most $1/\alpha$.

We calculate this expectation based on Theorem 2 of [9], which asserts that the expected image size of f^i is $(1-\tau_i)2^\ell$, where $\tau_0 = 0$ and $\tau_{i+1} = e^{-1+\tau_i}$. We show by induction on $i \geq 1$ that $\tau_i > 1-(2/i)$, which implies that the expected image size of f^i is less than $2 \cdot 2^\ell/i$, as required. The base is $i = 1$, for which indeed $\tau_1 = e^{-1} > -1$. Next, we need to prove that $\tau_{i+1} > 1-(2/(i+1))$, or $e^{-1+\tau_i} > 1-(2/(i+1))$. According to the assumption, it is sufficient to show that $e^{-2/i} > 1-(2/(i+1))$, or equivalently $-2/i > \ln(1-(2/(i+1)))$. This inequality can be deduced from the inequality $\ln(1+x) < x/(x/2+1)$ for $-1 < x < 0$ (see inequality (2) in [20]), by plugging in $x = -2/(i+1)$. □

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

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 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 settings.

As previously noted, when $2s + t \leq \ell$ each evaluated chain is not expected to collide with more than one different chain, and the pairs of chains can essentially be analyzed independently. Given a collision $\hat{x}$, We denote by A the event that 2 new chains of length 2^s, starting from arbitrary points (y_0, y_0'), also collide on $\hat{x}$. Below, we show that $\Pr[A] = \Theta(2^{2(2s-\ell)})$. Denote by B the event that the chains starting from (y_0, y_0') collide (on $\hat{x}$ or any other point), then $\Pr[B] = \Theta(2^{2s-\ell})$. We are interested in calculating the conditional probability $\Pr[A|B] = \Pr[\hat{x} = \hat{y}]$, and we have $\Pr[A|B] = \Pr[A \cap B]/\Pr[B] = \Pr[A]/\Pr[B] = \Theta(2^{2(2s-\ell)-(2s-\ell)}) = \Theta(2^{2s-\ell})$, as required.

We are left to show that $\Pr[A] = \Theta(2^{2(2s-\ell)})$. 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), so that $f(x_i) = f(x'_i) = \hat{x}$, with $x_i \neq x'_i$. Fixing (x_0, x'_0), we now calculate $\Pr[A]$, namely the probability that 2 new chains of length 2^s, starting from arbitrary points (y_0, y'_0), also 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). Clearly, $\Pr[A] = O(2^{2(2s-\ell)})$, as all 4 chains are of length $O(2^s)$. Hence, to conclude the proof, we need to show that $\Pr[A] = \Omega(2^{2(2s-\ell)})$.

In order to simplify the proof, we assume in the following that $0.25 \cdot 2^s \leq i \leq 0.75 \cdot 2^s$ and denote this event by I. We have $\Pr[I] \approx 1/2$ since the offset of the collision $\hat{x}$ is roughly uniformly distributed in the interval $[0, 2^s]$. This can be shown using Lemma 1, as increasing the length of the chains increases the collision probability (at a common offset) by the same multiplicative factor.

We have $\Pr[A|I] = \Theta(2^{2(2s-\ell)})$, since when $0.25 \cdot 2^s \leq i \leq 0.75 \cdot 2^s$, all 4 chains $(y_0, y_1, \ldots, y_{2^s-i},\ x_0, x_1, \ldots, x_i,\ y'_0, y'_1, \ldots, y'_{2^s-i},\ x'_0, x'_1, \ldots, x'_i)$ are of length $\Theta(2^s)$. Therefore, $\Pr[A] = \Pr[A|I] \cdot \Pr[I] + \Pr[A|\neg I] \cdot \Pr[\neg I] \geq \Pr[A|I] \cdot \Pr[I] = \Omega(2^{2(2s-\ell)})$ 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 2^t chains of fixed length 2^s, with **independent ℓ-bit random functions f_i**, such that $s + t < \ell$. Then $\Pr[(\hat{x}, i) = (\hat{y}, j)] = \Theta(2^{s-\ell})$. Furthermore, when $i = j$, we have $\Pr[\hat{x} = \hat{y}] = \Theta(2^{2s-\ell})$.*

Proof The proof follows essentially the same line of arguments as the proof of Lemma 3. We fix the collision $\hat{x}$ and 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 $\Pr[A] = \Theta(2^{2(s-\ell)})$.[3] Thus, $\Pr[A|B] = \Pr[A \cap B]/\Pr[B] = \Pr[A]/\Pr[B] = \Theta(2^{s-\ell})$ as claimed.

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 $\Pr[A|B] = \Pr[A \cap B]/\Pr[B] = \Pr[A]/\Pr[B] = \Theta(2^{2s-\ell})$. □

Lemma 5 *Let $\hat{x}$ be the entry point of an arbitrary cycle found by the cycle search algorithm for a 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 the probability that the chain y_i finds the same cycle and entry point $\hat{x}$ is $\Theta(2^{2s-\ell})$,* i.e. $\Pr[\exists i', j' \mid y_{i'} \neq y_{j'} \text{ and } y_{i'+1} = y_{j'+1} = \hat{x}] = \Theta(2^{2s-\ell})$.

Proof We denote the starting point of the chain found by the cycle search algorithm which collides (cycles) on $\hat{x}$ by x_0, and the corresponding collision by (x_i, x_j), with $i < j$, $x_i \neq x_j$ and $f(x_i) = f(x_j) = \hat{x}$. In the following, we assume that $0.25 \cdot 2^s \leq i, j \leq 0.75 \cdot 2^s$, which occurs with constant

[3] Once again, a lower bound on $\Pr[A]$ is easy to calculate assuming that $0.25 \cdot 2^s \leq i \leq 0.75 \cdot 2^s$ (which occurs with probability of about 1/2).

probability. The chain $\overrightarrow{y}$ will cycle with entry point $\hat{x}$ if $y_0, y_1, \ldots, y_{2^s-j}$ collides with $x_0, x_1, \ldots, x_i$. This occurs with probability of about $2^{2s-\ell}$ (when $0.25 \cdot 2^s \leq i, j \leq 0.75 \cdot 2^s$, the two chains are of length $\Theta(2^s)$), which proves that $\Pr[\exists i', j' \mid y_{i'} \neq y_{j'} \text{ and } y_{i'+1} = y_{j'+1} = \hat{x}] = \Omega(2^{2s-\ell})$. On the other hand, the probability that two chains of length at most 2^s collide is at most (roughly) $2^{2s-\ell}$, and therefore $\Pr[\exists i', j' \mid y_{i'} \neq y_{j'} \text{ and } y_{i'+1} = y_{j'+1} = \hat{x}] = \Theta(2^{2s-\ell})$. □

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 by the MAC oracle.

4.1 Collision filters

A simple collision filter was introduced in [19], and is also used in our work. A collision filter for an offline state x is a pair of message blocks (b, b'), with $b \neq b'$, such that we obtain the same state after processing these blocks from x (*i.e.* $h(x, b) = h(x, b')$). In order to build the filter, we find a collision in the underlying hash function by evaluating its compression function with state x, and about $2^{\ell/2}$ different messages blocks b. We apply the filter online on an unknown node x' — obtained after processing a message m' — by checking whether the tags of $m' \| b$ and $m' \| b'$ collide. If the state obtained after processing m' is not x, the tags of $m' \| b$ and $m' \| b'$ collide with probability only $2^{-n} < 2^{-\ell}$; therefore the collision filter identifies the state x with high probability.

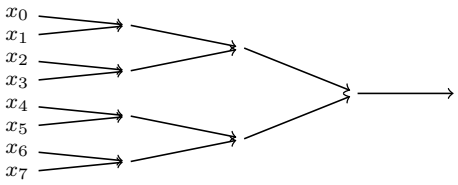
The complexity of building a collision filter offline is $O(2^{\ell/2})$. Testing the filter online requires querying the MAC oracle with $m' \| b$ and $m' \| b'$; assuming that the length of m' is $2^{s'}$, this requires $O(2^{s'})$ time.

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 [15]. 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 matching 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 by a factor about 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 [17], the time complexity of building the structure is $\Theta(2^{(\ell+t)/2})$. Note that for a HAIFA hash function, the nodes of each layer in the structure must be built using the same function and therefore they must have the same offset.



Once we finish building the diamond structure, we construct a standard collision filter for the final node y, using 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.

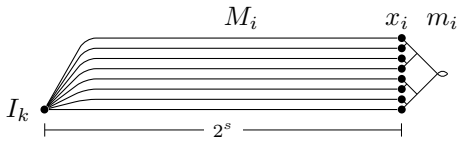
In order to test the filter for a state x_i (in the first layer of the diamond structure) on the unknown node x' obtained after processing a message m' online, we simply check whether the tags of $m' \parallel m_i \parallel [b]$ and $m' \parallel m_i \parallel [b']$ collide. Assuming that the length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

4.2.1 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 that 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 (both for HAIFA and Merkle-Damgård hash functions). Assuming that the messages M_i are of length 2^s, building 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.



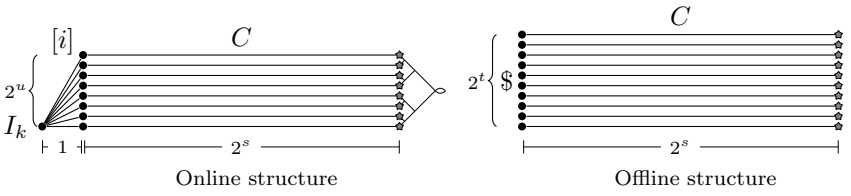
5 Internal State-recovery for HMAC with HAIFA

In this section, we describe the first internal state-recovery attack applicable to HAIFA. Our 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 roughly $2^{4\ell/5}$. We note that attacks against HAIFA can also be used to attack a Merkle-Damgård hash function, giving more freedom in the queried messages by removing the requirement for long sequences of identical blocks as in [19].

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 $M_i = [i] \,\|\, C$. 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'.



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:

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

The lowest complexity is reached when all the steps of the attack have the same complexity with $s = u = \ell/5$. More generally, we assume that $s \leq \ell/5$ and set $u = s$ to balance steps 2 and 3. This gives 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

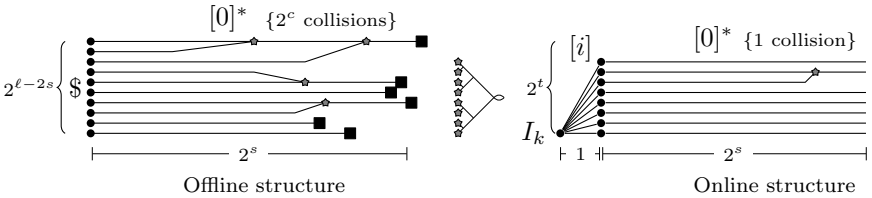
In this section, we revisit the results of [19], and give more flexible tradeoffs for various message lengths.

6.1 Tradeoff 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 [19] by using two improved techniques. First, while [19] used a same-offset offline collision search, we use a more general free-offset offline collision search which enables us to find collisions more efficiently. Secondly, while [19] used collision filters, we use a more efficient diamond filter.

Attack 2: Chain-based tradeoff 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 among the tags.
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 $\hat{y} = \hat{x}$.



We generate collisions offline using free-offset collision search, build a diamond filter for the collision points ($\star$), 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 same-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:	$2^{\ell-s}$	**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:	$2^{c+s} = 2^{\ell-s}$		

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 Tradeoff based on cycles

We now generalize the cycle-based state-recovery attack of [19] which 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). The attack of [19] uses messages of length $2^{\ell/2}$ and has a complexity of $2^{\ell/2}$.

Our attack uses the same idea of [19], but searches for (potentially) shorter cycles using (potentially) shorter messages of length 2^s for $s \leq \ell/2$. The complexity of the attack $2^{2\ell-3s}$.

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

1. (offline) Search for a cycle in the functional graph of $f = h_{[0]}$, 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 $\hat{x}$. Build a collision filter for $\hat{x}$.
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 $M' =$

$[b] \| [0]^{2^s+L} \| [1] \| [0]^{2^s}$ (both of length $1+2^s+1+2^s+L = 2+2^{s+1}+L$), until $\mathrm{MAC}(M) = \mathrm{MAC}(M')$.

3. (online) Define $M_i = [b] \| [0]^i \| [1] \| [0]^{2^s+L}$ and $M'_i = [b] \| [0]^{i+L} \| [1] \| [0]^{2^s}$, and use a binary search to find the minimum value of i such that M_i and M'_i have the same MAC. Use the collision filter to test whether the state reached after $[b] \| [0]^i$ is equal to $\hat{x}$.

Complexity analysis. The intuition behind this attack is the same as the cycle-based attack from [19]. Step 2 will be successful if the message $M_0 = [b] \| [0]^{2^s} \| [1] \| [0]^{2^s}$ reaches the cycle of length L twice: once after processing $[b] \| [0]^{2^s}$ and again after processing the full message M'. Indeed, if this is the case, then adding L zero blocks in the first cycle (in message M') or in the second cycle (in message M) does not change the state reached before the finalization function. In addition, M and M' have the same length $2+2^{s+1}+L$ which allows the collision to be propagated to the output (a simple attack that enters the cycle only once would not work because of the different lengths).

More formally, we need the two following conditions to be satisfied. First, the states obtained after evaluating the prefixes $[b] \| [0]^{2^s}$ and $[b] \| [0]^{2^s+L}$ must collide. This occurs if one of the states computed during the evaluation of $[b] \| [0]^{2^s}$ collides with $\hat{x}$ (and thus enters the cycle of length L), which has probability $\Theta(2^{2s-\ell})$ according to Lemma 5. Secondly, the states obtained after evaluating the suffixes $[1] \| [0]^{2^s+L}$ and $[1] \| [0]^{2^s}$ must also collide. Similarly to the previous case, this occurs if one of the states computed during the evaluation of $[1] \| [0]^{2^s}$ collides with $\hat{x}$. Again, this event occurs with probability $\Theta(2^{2s-\ell})$ according to Lemma 5. Thus, the success probability of Step 2 is $\Omega(2^{2(2s-\ell)})$ (if fact, it is $\Theta(2^{2(2s-\ell)})$), 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 $O(2^{2(\ell-2s)+s}) = O(2^{2\ell-3s})$.

Step 3 will detect the minimum i such that M_i reaches the cycle, therefore the i-th state is the entry point of the cycle. Step 3 is expected to be run once for the correct pair, but each block value b can lead to an internal collision (at a common offset) without reaching the cycle with probability about $2^{s-\ell}$, according to Lemma 1. Thus, Step 3 is run $2^{2(\ell-2s)+s-\ell} = 2^{\ell-3s}$ times with false positives.

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

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

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

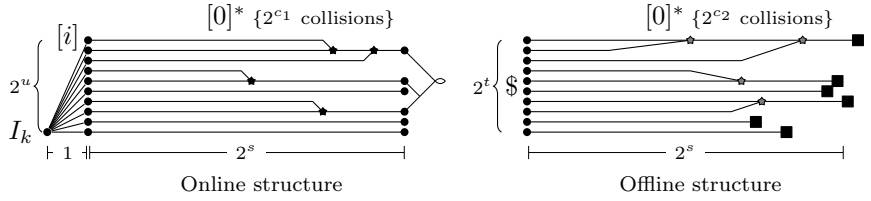
message length of the underlying hash function is very restricted (*e.g.* the SHA-2 family), and moreover they play an important role in the key-recovery attack on HMAC with the GOST R 34.11-2012 hash function (described in Section 8). In order to reach a complexity of $2^{\ell-2s}$, we combine the idea of building filters in the online phase with the use of collisions as special states (using the results of Lemma 3 for Merkle-Damgård and Lemma 4 for HAIFA).

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.

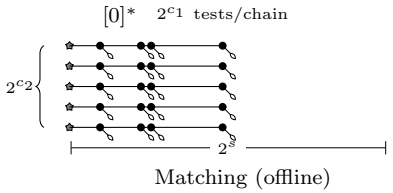
7.1 Merkle-Damgård

Attack 4: 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) d_{ij} from the starting point to the collision, and denote the (unknown) state reached 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 $2^s - d_{ij}$ (*i.e.*, $f^{2^s-d_{ij}}(\hat{x})$) with y_{ij} using the diamond filter. If a match is found, then with high probability $y_{ij} = f^{2^s-d_{ij}}(\hat{x})$.



We generate collisions, build an online diamond filter for their endpoints, and match them with iterates of collisions found offline.



Complexity analysis. Using similar analysis to Section 6.1, we have $c_1 = 2u + s - \ell$ (as a pair of chains collides 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 $c_2 = \ell/2$, implying that $c_1 = \ell/2 - 2s$. 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 $\tilde{O}(2^{\ell-2s})$.

We note that the complexity can be reduced to a minimum of $O(2^{3\ell/4})$ (without logarithmic factors) by using messages which are slightly longer than $2^{\ell/8}$. In particular, this gives optimal attacks for functions of the `SHA-2` family.

7.2 HAIFA

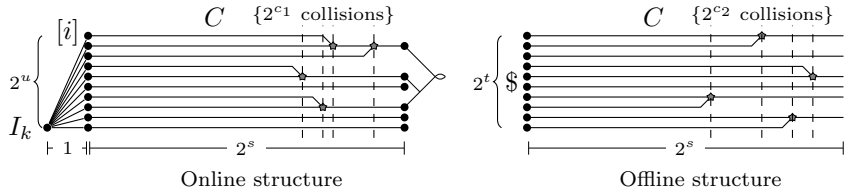
The general structure of the attack on HAIFA is similar to one of the previous attack on Merkle-Damgård. The main difference between the attacks is that we are forced to use same-offset collision search offline, rather than free-offset collision search as in the previous attack.

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

1. (online) Fix an arbitrary message suffix C of length 2^s, query the oracle with 2^u messages $M_i = [i] \parallel C$, and locate 2^{c_1} collisions.
2. (online) For each collision (i, j), use a binary search to find the distance (offset) d_{ij} from the starting point to the collision, and denote the (unknown) state reached 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 same-offset collision search algorithm by computing the compression function with the message C from 2^t arbitrary starting points, and locate 2^{c_2} collisions.
4. (offline) For each offline collision $\hat{x}$, match the endpoint of its chain with all online collisions $y_{ij} \in Y$ that occur at the same offset as $\hat{x}$. More precisely, for each $\hat{x}$ with offset d, use the diamond filter to test only the endpoint of its chain (at offset 2^s) with the corresponding collisions in Y that occur at offset $d_{ij} = d$. If a match with some y_{ij} is found, then with high probability the state reached after M_i and M_j is the endpoint of the offline collision $\hat{x}$.



We generate collisions and build an online diamond filter, and match the endpoints with offline collisions endpoints 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 occur 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. Similarly to the analysis of the previous attack on Merkle-Damgård, we have $c_1 = 2u + s - \ell$, but as we use same-offset collision search offline, we can detect a smaller number of $c_2 = 2t + s - \ell$ collisions. We note that the online collision offsets d_{ij} are essentially uniform (see Lemma 1), and therefore in Step 4, each offline collision $\hat{x}$ at offset d is matched with about $2^{c_1} \cdot 2^{-s}$ states in Y.

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:**	$2^{c_1+c_2-s} \cdot c_1 = 2^{\ell-2s} \cdot c_1$

The optimal tradeoffs are achieved by balancing steps 2 and 3, *i.e.* with $c_2 = \ell/2$, implying that $c_1 = \ell/2 - s$. 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:**	$2^{\ell-2s} \cdot \ell/2$

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 Key-Recovery Attack on HMAC with GOST R 34.11-2012 (Streebog)

In this section we are interested in hash functions that use an internal checksum, as shown in Figure 4. In [19], the state-recovery attack on HMAC with a Merkle-Damgård hash function was extended into a key-recovery attack, in case the hash function uses an internal checksum like the GOST R 34.11-94 hash function. Here, 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 $2^{4\ell/5}$) can be extended into a key-recovery attack (with complexity $2^{4\ell/5}$).

In particular, this attack is applicable to the standard GOST R 34.11-2012 (aka Streebog), and gives a key-recovery attack with complexity 2^{417} for its 512-bit version. This result shows that HMAC-GOST R 34.11-2012 is weaker than HMAC-SHA-512 (for which no key-recovery attack better than exhaustive search is known).

8.1 Description of the attack

The attack uses the same framework as [19], exploiting the structure of hash functions with a checksum. We target the finalization function in the first hash function call. The input state value $x_\star$ can be recovered using the previous state-recovery attacks, and we exploit the fact that the checksum value is key dependant, but can be controlled by injecting differences in the message: $\sigma = K \oplus \texttt{ipad} \oplus \texttt{Sum}^{\oplus}(M)$. 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 such collisions offline since $x_\star$ and L are known. Moreover, if we find two collisions with the same difference in the σ input, there is a high probability that the actual σ values are the same, because the other inputs to g ($x_\star$ and L) are fixed (on average, we expect a single collision with a fixed difference). Once we find an online collision and an offline collision with the same difference, we can therefore recover the value of K by exploiting the equation $\sigma = K \oplus \texttt{ipad} \oplus \texttt{Sum}^{\oplus}(M)$.

8.2 Detailed attack process

The first step of the attack uses the state-recovery attack of Section 7.2. However, unlike hash functions without checksums, equal tags of arbitrary messages of the same length (but with different checksums) do not imply equality (with high probability) of the states reached after the messages are processed. Therefore, we execute Attack 5 with the modifications below that ensure we only compare tags of messages with a fixed checksum.

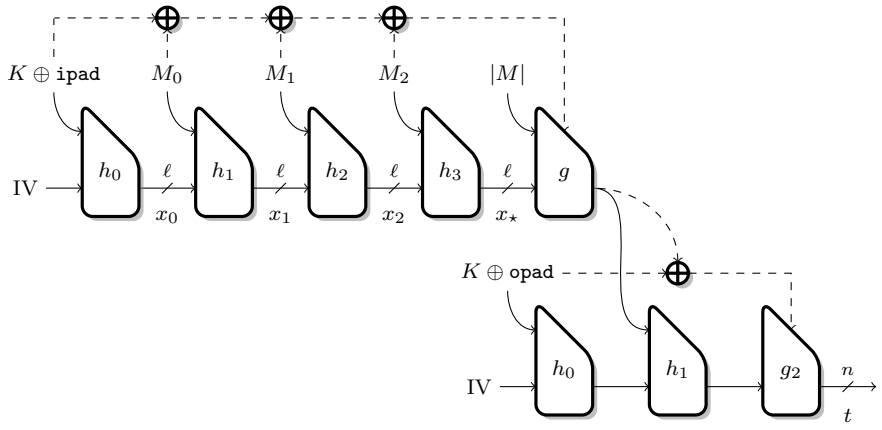


Fig. 4 `HMAC` based on a hash function with a block counter and a checksum (dashed lines).

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

Attack 6: Key-recovery attack against `HMAC` with a `GOST`-like hash function
Complexity $\tilde{O}(2^{4\ell/5})$

0. Execute Attack 5 to recover a state x_1 obtained after processing some message M_1, with $|M_1| = s = \ell/10$.
1. (offline) Starting from state x_1, use Joux's multicollision attack [14] to generate a set of $2^{7\ell/10}$ messages that all collide on an 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 MAC oracle with the set of messages from Step 1 and collect collisions ($2^{2 \cdot 7\ell/10 - \ell} = 2^{2\ell/5}$ collisions are expected). For each collision (M, M'), compute the checksum difference $\Delta M = \texttt{Sum}^{\oplus}(M) \oplus \texttt{Sum}^{\oplus}(M')$, and store $(\Delta M, \texttt{Sum}^{\oplus}(M))$.
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 2. If a match is found, consider $\texttt{Sum}^{\oplus}(M) \oplus \sigma \oplus \texttt{ipad}$ and $\texttt{Sum}^{\oplus}(M) \oplus \sigma' \oplus \texttt{ipad}$ as potential key candidates, and test them using a known tag.

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

Complexity.

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

The total complexity is $\tilde{O}(2^{4\ell/5})$. In Appendix B, we give a more precise complexity evaluation of this attack and show that it requires less than 2^{417} operations on average when applied to `Streebog` (with $\ell = 512$).

Significance of short message attacks. The efficient state-recovery attack for short messages is an important factor here, as the message length directly influences the time complexity of the attack in Step 2. In fact, state-recovery attacks with complexity $2^{\ell-s}$ as in Section 5 (or in [19], for the Merkle-Damgård construction) can only reach complexity $2^{5\ell/6}$ for key-recovery, while the attacks with complexity $2^{\ell-2s}$ (described in Section 7) allow to reach complexity $2^{4\ell/5}$.

We note that in `GOST` R 34.11-94 (which is based on Merkle-Damgård as opposed to HAIFA) the message length is processed with the same function as the other blocks. This (in addition to some padding properties) allowed the attack of [19] to deduce the state of a short message from the state of a long message. Therefore, unlike our key-recovery attack on `HMAC` with `GOST` R 34.11-2012, efficient state-recovery attacks for short messages did not play an important role in the key-recovery attack on `HMAC` with `GOST` R 34.11-94 [19].

9 Universal Forgery Attacks with Short Queries

In the setting of universal forgery attacks the adversary receives a challenge message C at the beginning of the game. The goal of the adversary is to predict the tag of the challenge by interacting with the MAC oracle, but without querying for the actual challenge C.

We now revisit the universal forgery attack of Peyrin and Wang [24]. This attack has two phases, where in the first phase the adversary recovers one of the internal states computed by the MAC oracle on the challenge C. In the second phase, the adversary uses a second-preimage attack on long messages in order to generate a different message C' known to have the same tag as the challenge. Thus, by querying the MAC oracle for C', the adversary can forge the tag of C. The first phase of the attack is the most expensive; the attack of [24] requires $\tilde{O}(2^{5\ell/6})$ computations, which has been improved to $\tilde{O}(2^{3\ell/4})$ in [11] using a heuristic assumption on the functional graph of random functions.

The main drawback of those attacks is that its first phase uses very long queries to the MAC oracle, regardless of the length of the challenge.[4] Therefore,

[4] The first phase of the attacks of [11,24] uses queries of length $2^{\ell/2}$. Their algorithms are related to the cycle-based attack of [19] that is generalized is Section 6.2.

the attack is inapplicable to many concrete HMAC instantiations where the hash function limits the message length (such as HMAC-SHA-1 and HMAC-SHA-2).

In this section, we use the tools developed in this paper to devise two universal forgery attacks using shorter queries to the MAC oracle. For a challenge of length 2^s, our first universal forgery attack has a complexity of $\tilde{O}(2^{\ell-s})$ with $s \leq \ell/7$, using queries to the MAC oracle of length of at most 2^{2s} (which is significantly smaller than $2^{\ell/2}$ for any $s \leq \ell/7$). Thus, the optimal complexity of this attack is $\tilde{O}(2^{6\ell/7})$, obtained with a challenge of length at least $2^{\ell/7}$ and queries of length $2^{2\ell/7}$. Our second universal forgery attack has a complexity of only $\tilde{O}(2^{\ell-s/2})$. However, it is applicable for any $s \leq 2\ell/5$, using queries to the MAC oracle of length of at most 2^s. Thus, this attack has an improved optimal complexity of $\tilde{O}(2^{4\ell/5})$, which is obtained with a challenge of length at least $2^{2\ell/5}$ and queries of length $2^{2\ell/5}$.

In order to devise our attacks, we construct new state-recovery algorithms, but reuse the second phase from [24] (*i.e.*, the second-preimage attack) in both of the attacks. Thus, in the following, we concentrate on the state-recovery algorithms. For the sake of completeness, we describe the second phase of the attack in Appendix A. Since its complexity is about $2^{\ell-s}$ for any $s \leq \ell/2$, it does not add a significant factor to the total time complexity.

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

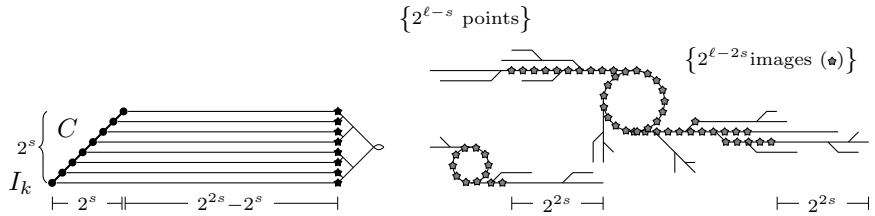
Directly matching the 2^s states computed by the MAC oracle on the challenge message with some states evaluated offline is too expensive. Therefore, we first compute and match the images of the states under iterations of a fixed function (as images match with higher probability than arbitrary states). Then, we use the initial matching of the images to efficiently match and recover an actual state that is computed by the MAC oracle on the challenge message. The 2-phase matching used in the attack resembles the approach used in [11,24], but the main difference is that the first phase of the matching does not exploit cycles in the functional graph. Moreover, the second phase of the matching is performed using an efficient binary search matching algorithm (Algorithm 1, whose pseudo-code is give in Appendix C). This algorithm can be simplified using a heuristic assumption on the number of nodes that need to be matched in the second phase (such an assumption was made explicitly in [11], and more implicitly in [6]). However, the analysis of Algorithm 1 is more rigorous and does not require any conjecture.

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

Attack 7: Universal forgery attack using chains (first phase)
Complexity $\tilde{O}(2^{\ell-s})$, with $s \leq \ell/7$ (min: $2^{6\ell/7}$)
Query length: $O(2^{2s})$

1. (online) Query the oracle with 2^s messages $M_i = C_{|i} \,\|\, [0]^{2^{2s}-i}$. Denote the set of (unknown) final states of the chains by Y. Build an online diamond filter for all states in Y.
2. (offline) Compute a structure of chains containing a total of $2^{\ell-s}$ points. Each chain is extended until it cycles or collides with a previous chain. Consider the set X of the 2^{2s}-iterates of f (namely images of $f^{2^{2s}}$ in the structure). According to Lemma 2, this set contains (no more than) about $2^{\ell-2s}$ distinct points. Build an online diamond filter for X.
3. (offline) Match all the points $x \in X$ with the 2^s points in Y.
4. (offline) For each match between $x \in X$ and an online state in Y (obtained using M_i), use an additional matching algorithm to test the actual message $C_{|i}$: call the binary search matching algorithm (Algorithm 1) with:
   - input message $M_i = C_{|i} \,\|\, [0]^{2^{2s}-i}$;
   - the tree rooted at x (obtained by disconnecting the edge between x and $f(x)$ from the graph and considering all the points that merge into x);
   - and distance $2^{2s} - i$.

   If the algorithm returns a match y', then with high probability the state obtained after processing $C_{|i}$ is equal to y'.



We efficiently detect a match between the challenge points ($\bullet$) and the offline structure, by first matching X ($\star$) and Y ($\star$).

Algorithm 1: Binary search matching
Inputs: distance d, message M of length $d' > d$ whose last d blocks are zero, tree of chains computed with the zero block that merge into the root x
Output: node y in the tree with distance d to x that is equal to the state reached after evaluating $M_{|d'-d}$ (NULL if y does not exist)

1. Denote by $\textsc{Size}(u)$ the number of nodes in the tree whose root is u. Traverse the tree rooted at x (backwards) for at most d steps until a leaf or a collision is encountered:

- If a total of d steps were traversed, denote the node at distance d from x by z. Build a collision filter for z and test it on $M_{|d'-d}$. If they match, return $y = z$, otherwise go to Step 2.
- If a leaf is encountered (before d steps were traversed), denote the current node by z and go to Step 2.
- If colliding states z_1, z_2 such that $f(z_1) = f(z_2) = z$ are encountered, then if $\textsc{Size}(z_1) \geq \textsc{Size}(x)/2$ continue traversing z_1 in Step 1. Otherwise, if $\textsc{Size}(z_2) \geq \textsc{Size}(x)/2$ continue traversing z_2 in Step 1.[a]
  Otherwise (sizes of trees rooted at z_1 and z_2 are small), let $d(z)$ denote the distance of z from x. Build collision filters for z_1 and z_2 and test them on $M_{|d'-d(z_1)}$ (note that $d(z_1) = d(z_2) = d(z) + 1$). If there is no match, go to Step 2. Otherwise, assume without loss of generality that the match is with z_1. Call the algorithm recursively with input message $M_{|d'-d(z_1)}$, tree rooted at z_1, and distance $d' - d(z_1)$ (note that $\textsc{Size}(z_1) < \textsc{Size}(x)/2$).

2. Backtrack (by cutting off half of the tree): Denote by z the current node in the traversal. Let z' be the first ancestor of z (in the direction of the root x) for which there is a collision $f(z') = f(z'')$ for some z''. If there is no such z', return NULL. Otherwise, call the algorithm recursively with input message M, tree rooted at x with z' cut off, and distance d (note that $\textsc{Size}(z') \geq \textsc{Size}(x)/2$).

[a] The algorithm can be easily generalized to deal with collisions between $t \geq 2$ states by analyzing all t nodes in the t-way collision (looking for a child whose subtree size is at least half of the total tree size, and building filters for all the children if required). As the maximal value of t in an ℓ-bit random mapping is not expected to exceed ℓ, such collisions only add (up to) a logarithmic factor to the total complexity of the algorithm.

Analysis. The offline structure of Step 2 contains $2^{\ell-s}$ distinct points, and thus according to the birthday paradox, it covers one of the 2^s points of the challenge with high probability. In this case, the attack will successfully recover the state of the covered point with high probability, since the 2^{2s}-iterate of the covered point is also covered by the offline structure and will be matched in Step 3.[5]

The analysis of Algorithm 1 used in Step 4 is based on the following property: the size of the tree parameter is cut by at least half in every recursive call, where in each such recursive call we traverse at most $d \leq 2^{2s}$ nodes and compute and test a small constant number of collision filters. Since the initial tree has at

[5] The only case in which the 2^{2s}-iterate (denoted by x) of a covered point y is not covered by the offline structure, is when this iterate goes through the disconnected edge from x to $f(x)$ (namely, $f(x)$ is on a path from y to x whose length is at most 2^{2s}). This can only occur if x is on a cycle with less than 2^{2s} elements. Such (small) cycles are expected to belong to (small) connected components whose total size is $O(2^{2\cdot 2s}) = O(2^{4s})$ (see [9]). On the other hand, we evaluate a total of $2^{\ell-s}$ nodes offline, and since $2^{4s} \ll 2^{\ell-s}$ for $s \leq \ell/7$, it is unexpected that the covered point y will belong to such a small component.

most 2^ℓ nodes, there are at most ℓ recursive calls in Algorithm 1 (for a single call in Step 4). The complexity of each recursive call is dominated by building a few offline collision filters, and therefore the complexity of Algorithm 1 is about $\ell \cdot 2^{\ell/2}$ (note that testing the filter online has complexity $2^{2s} < 2^{\ell/2}$, and traversing nodes in each recursive call has complexity at most $d \leq 2^{2s} < 2^{\ell/2}$).

Since we have 2^s points in Y and all the points in X are distinct, there are at most 2^s matches in Step 3, and the time complexity of Step 4 is at most $\ell \cdot 2^{\ell/2} \cdot 2^s = \ell \cdot 2^{\ell/2+s}$.

Complexity.

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

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

9.2 A universal forgery attack using collisions

In this attack, we devise a different algorithm which recovers one of the states computed during the MAC computation on the challenge message. The main idea here is to begin with a state-recovery attack in order to determine the value of some online states. Then, we use iterates of a fixed function on the challenge states and on the known points, and detect (online) collisions between the two sets of iterates. This allows to determine the values of several iterates of challenge states, and removes the need for online filters (that are required in the first matching phase of the previous attack). Next, we evaluate some states offline and execute the second matching phase on offline iterates that match the online challenge iterates. This allow to recover an actual challenge state (the second phase matching uses Algorithm 1 as the previous attack).

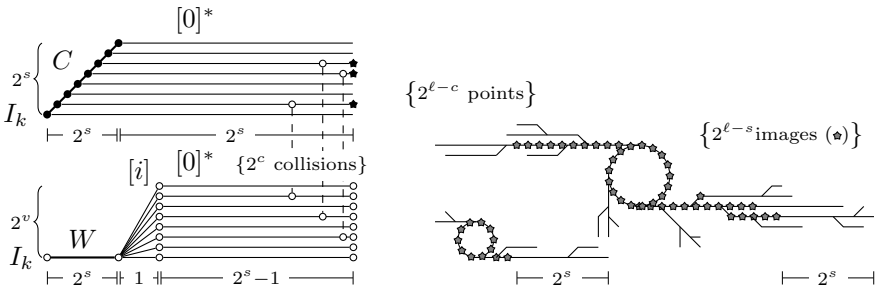
Attack 8: Universal forgery attack using collisions (first phase)
Complexity $O(2^{\ell-s/2})$, with $s \leq 2\ell/5$ (min: $2^{4\ell/5}$)
Query length: $O(2^s)$

1. (online) Query the oracle with 2^s messages $M_i = C_{|i} \parallel [0]^{2^{s+1}-i}$, and sort the tags.
2. (online) Execute state-recovery Attack 2 using messages of length $\min(2^s, 2^{\ell/3})$, and denote by W a message of length 2^s whose last computed state is recovered.[a]
3. (online) Query the oracle with 2^v messages $W_j = W \parallel [j] \parallel 0^{2^s-1}$, 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, the

state reached after M_i is known, because the state reached after W is already known. We denote the set of those states as X.

4. (offline) Compute a structure of chains containing a total of $2^{\ell-c}$ points. Each chain is extended until it cycles or collides with a previous chain.
5. (online) For each offline point in the structure y which matches a point in X (corresponding to M_i), call the binary search matching algorithm (Algorithm 1) with inputs:
   - message M_i;
   - the tree rooted at y (obtained by disconnecting the edge between y and $f(y)$ from the graph and considering all the points that merge into y);
   - and distance $2^{s+1} - i$.

   If the algorithm returns a match y', then with high probability the state obtained after processing $C_{|i}$ is equal to y'.



We match the known points in X (⋆) and Y (⋆) in order to detect a match between the challenge points (•) and the offline structure.

[a] In case $s > \ell/3$, 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^s.

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 such a prefix $C_{|i}$. Since iterate $2^{s+1} - i$ of $C_{|i}$ is also covered by the offline structure,[6] then the state corresponding to $C_{|i}$ will be recovered in Step 5.

In order to calculate the value of c, note that the online structure computed in Step 1 contains 2^s chains, each of length at least 2^s, and thus another arbitrary chain of length 2^s collides with one of the chains in this structure at the same offset with probability of about $2^{2s-\ell}$ (see Lemma 1). Since the

[6] This occurs with high probability, and can be shown by analyzing the cycle structure of the graph as in the previous attack.

structure computed in Step 3 contains 2^v such chains, the expected number of detected collisions between the structures is $2^c = 2^{2s+v-\ell}$, *i.e.*, $c = 2s + v - \ell$.

In Step 5, we call Algorithm 1 at most 2^c times (at most once for each online collision), and each invocation has complexity $\ell \cdot 2^{\ell/2}$ (the analysis of Algorithm 1 is essentially the same as in the previous attack). In total, the complexity of Step 5 is $\ell \cdot 2^{c+\ell/2} = \ell \cdot 2^{2s+v-\ell/2}$.

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

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

10 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 for `HMAC` with hash functions based on the HAIFA mode, and provided improved tradeoffs between the message length and the complexity of state-recovery attacks for `HMAC` with 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.

We give a more concrete application of some of these results in Section 8, with a key-recovery attack against `HMAC` with the new Russian standard `GOST` R 34.11-2012. In particular this attack requires a key-recovery attack against `HMAC` with HAIFA hash functions, and improved tradeoffs for short messages.

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 (perhaps under certain natural assumptions). At the same time, it would be useful to find additional applications of our algorithms to related cryptosystems (such as various modes of operation) where some level of security beyond the birthday bound is required.

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. Dinur, I., Leurent, G.: Improved Generic Attacks against Hash-Based MACs and HAIFA. In: Garay and Gennaro [10], pp. 149–168
7. Dolmatov, V., Degtyarev, A.: GOST R 34.11-2012: Hash Function. RFC 6986 (Informational) (Aug 2013), http://www.ietf.org/rfc/rfc6986.txt
8. 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
9. Flajolet, P., Odlyzko, A.M.: Random Mapping Statistics. In: Quisquater, J., Vandewalle, J. (eds.) Advances in Cryptology - EUROCRYPT '89, Workshop on the Theory and Application of of Cryptographic Techniques, Houthalen, Belgium, April 10-13, 1989, Proceedings. Lecture Notes in Computer Science, vol. 434, pp. 329–354. Springer (1989)
10. Garay, J.A., Gennaro, R. (eds.): Advances in Cryptology - CRYPTO 2014 - 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part I, Lecture Notes in Computer Science, vol. 8616. Springer (2014)
11. Guo, J., Peyrin, T., Sasaki, Y., Wang, L.: Updates on Generic Attacks against HMAC and NMAC. In: Garay and Gennaro [10], pp. 131–148
12. 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: Cid, C., Rechberger, C. (eds.) Fast Software Encryption - 21st International Workshop, FSE 2014, London, UK, March 3-5, 2014. Revised Selected Papers. Lecture Notes in Computer Science, vol. 8540, pp. 571–590. Springer (2014)
13. Guo, J., Sasaki, Y., Wang, L., Wu, S.: Cryptanalysis of HMAC/NMAC-Whirlpool. In: Sako and Sarkar [25], pp. 21–40
14. 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)
15. 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)
16. 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)
17. Kortelainen, T., Kortelainen, J.: On Diamond Structures and Trojan Message Attacks. In: Sako and Sarkar [25], pp. 524–539
18. Leurent, G., Peyrin, T., Wang, L.: New Generic Attacks Against Hash-based MACs. IACR Cryptology ePrint Archive 2014, 406 (2014), http://eprint.iacr.org/2014/406
19. Leurent, G., Peyrin, T., Wang, L.: New Generic Attacks against Hash-Based MACs. In: Sako and Sarkar [25], pp. 1–20
20. Love, E.R.: Some Logarithm Inequalities. The Mathematical Gazette 64(427), 55–57 (1980), http://www.jstor.org/stable/3615890
21. 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)
22. van Oorschot, P.C., Wiener, M.J.: Parallel Collision Search with Cryptanalytic Applications. J. Cryptology 12(1), 1–28 (1999)
23. 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)

24. Peyrin, T., Wang, L.: Generic Universal Forgery Attack on Iterative Hash-Based MACs. In: Nguyen, P.Q., Oswald, E. (eds.) Advances in Cryptology - EUROCRYPT 2014 - 33rd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Copenhagen, Denmark, May 11-15, 2014. Proceedings. Lecture Notes in Computer Science, vol. 8441, pp. 147–164. Springer (2014)
25. 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)
26. Tsudik, G.: Message authentication with one-way hash functions. SIGCOMM Comput. Commun. Rev. 22(5), 29–38 (Oct 1992)
27. 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 The Second Phase of the Universal Forgery Attacks [24]

In this section we describe the second phase of our universal forgery attacks, which is identical to the second phase of the attack of [24]. Recall the we are given a challenge message $C = m_1 \| m_2 \| \ldots \| m_{2^s}$ of length 2^s blocks and our goal is to predict the tag of C by querying the MAC oracle with queries that are different from C. We denote the internal states computed by the MAC oracle on the challenge C by $y_0, y_1, \ldots, y_{2^s}$. In the first phase of the attack, we recover one of these internal states, which we denote by y_p. We assume that $p \leq 2^{s-1}$. This can be assured by running the first phase of the attack only on the first half of the blocks of C, which results in a small constant penalty in the complexity of the attack.

In the second phase of the attack we use a second-preimage attack on long messages in order to generate a different message C' which has the same tag as the challenge. The second-preimage attack is based on the one by Kelsey and Schneier [16]. The attack uses a structure known as an (i, j)-expandable message (for integers $i < j$), proposed in [16]. An (i, j)-expandable message is composed of an arbitrary initial state x and a final state z where for each integer d such that $i \leq d \leq j$, there is an efficiently computable message of d blocks $M_d = m'_1 \| m'_2 \| \ldots \| m'_d$ that links x to z, namely, $z = h(\ldots h(h(x, m'_1), m'_2) \ldots, m'_d)$. Assuming that $i \leq l/2$ (which is the case in our attack), an $(i, 2^i)$-expandable message can be constructed from any initial state x in complexity of $\ell \cdot 2^{\ell/2}$. The details of how to construct this structure are found in [16].

Algorithm 2: Universal Forgery Attack (second phase)
Complexity $\tilde{O}(2^{\ell-s})$, with $s \leq \ell/2$

1. (offline) Compute the states $y_p, y_{p+1}, \ldots, y_{p+2^{s-1}}$ and store them in a sorted table.
2. (offline) Build an $(s-1, 2^{s-1})$-expandable message starting from y_p and denote its final state by z.
3. (offline) For about $2^{\ell-s}$ values of the single-block $[i]$, compute $h(z, [i])$ and search the sorted table for y_j such that $y_j = h(z, [i])$ and $j \geq p + s$. Once such i and j are found, continue to the next step.
4. (online) Let M_{j-1-p} be the message of $j-1-p$ blocks that links y_p to z, computed from the expandable message. Let $C' = C_p \| M_{j-p-1} \| [i] \| m_{j+1} \| m_{j+2} \| \ldots \| m_{2^s}$ be the 2^s-block message computed by concatenating the p-block prefix of C with $M_{j-p-1} \| [i]$ and the suffix of $2^s - j$ blocks of C. Query the MAC oracle with C' and denote the result by T.
5. (offline) Output T as the tag for C.

Complexity.

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

For $s \leq \ell/2$ the total complexity is $\tilde{O}(2^{\ell-s})$ as required.

B Concrete Complexity of HMAC-Streebog Key-Recovery

In this section we calculate with better precision (taking into account constants and logarithmic factors) an upper bound on the expected complexity of Attack 6 (applied to Streebog with $\ell = 512$). While the analysis can be refined to give slightly better complexity, our calculations demonstrate that the constants involved in our attacks are indeed small.

Attack 6 is dominated by Step 0, which uses Attack 5, with $s = \ell/10$. Therefore, we start with a concrete analysis of Attack 5.

Attack 5. The most expensive steps are 2 and 4, where we match two sets by building and using a diamond filter. In order to optimize the attack, we use parameters $c_1 = \ell/2 - s$ and $c_2 = \ell/2 + \alpha$ for a small constant α (*e.g.* $\alpha = 4$), and run steps 1 and 3 until we have the required number of collisions (2^{c_1} and 2^{c_2}).

The expected value of u required to generate 2^{c_1} collisions in Step 1 is roughly $u = (c_1 + \ell - s + 1)/2$, accounting for the factor $\frac{1}{2}$ in the number of message pairs. Similarly, the expected value of t in Step 3 is $t = (c_2 + \ell - s + 1)/2$. This results in a complexity that is still negligible compared to steps 2 and 4, in spite of the small constants added ($2^{3\ell/4+1/2}$ for Step 1 and $2^{3\ell/4+(s+\alpha+1)/2}$ for Step 3).

With these parameters the online diamond filter for the set of size 2^{c_1} will have depth $\lceil c_1 \rceil$, and require less than $8\frac{e}{e-1}2^{c_1/1+\ell/2}$ evaluations of the MAC (following [17]). Since each query has 2^s blocks, the total complexity is $8\frac{e}{e-1}2^{s+c_1/2+\ell/2} < 2^{3\ell/4+s/2+3.7}$ for Step 2.

Finally, in Step 4, we use only collisions with an offset d such that $d \geq 2^{s-1}$, to simplify the analysis (we keep about $2^{c_2-1} = 2^{\ell/2+3}$ offline collisions). Following Lemma 4, we fix an offline collision $\hat{x}$ (with preimages x_d and x'_d) and we evaluate the probability that a random pair of chains y_0, y'_0 reaches the same collision, under the condition that y_0, y'_0 collide at the same offset d. Let A denotes the event that y_0, y'_0 collide on $\hat{x}$ and B the event that they collide at offset d on an arbitrary point (all collisions are same-offset collisions). We first evaluate the probability that the chains starting at y_0 and y'_0 reach x_d or x'_d, and then the probability that y_0, y'_0 collide on $\hat{x}$ (namely $\Pr[A]$).

$$\Pr[y_0 \not\rightsquigarrow x_d] \leq (1 - 2^{-\ell})^{2^{s-1}} \leq e^{-2^{s-1-\ell}}$$

$$\Pr[y_0 \rightsquigarrow x_d] = 1 - \Pr[y_0 \not\rightsquigarrow x_d] \geq 1 - e^{-2^{s-1-\ell}} \approx 2^{s-1-\ell}$$

$$\Pr[A] \geq \Pr[y_0 \rightsquigarrow x_d] \cdot \Pr[y'_0 \rightsquigarrow x'_d] + \Pr[y'_0 \rightsquigarrow x_d] \cdot \Pr[y_0 \rightsquigarrow x'_d] \geq 2^{2s-2\ell-1}$$

We have $\Pr[B] \leq 2^{-\ell}$, hence $\Pr[A|B] = \Pr[A \cap B]/\Pr[B] \geq 2^{2s-\ell-1}$. Therefore, the matching of Step 4 is expected to succeed after less than $2^{\ell-2s+1}$ attempts on average (the actual complexity without the restriction $d \geq 2^{s-1}$ is slightly smaller). Since each match requires to follow a path in a diamond of length c_1, the total complexity is less than $c_1 \cdot 2^{\ell-2s+1} \leq \ell \cdot 2^{\ell-2s}$ on average. This results in the following complexities:

Step 1:	$2^{3\ell/4+1/2}$	**Step 2:**	$2^{3\ell/4+s/2+3.7}$
Step 3:	$2^{3\ell/4+s/2+(\alpha+1)/2}$	**Step 4:**	$\ell \cdot 2^{\ell-2s}$

In particular, with $\ell = 512$ and $s = 54$, we obtain a complexity of $2^{415.1}$.

Attack 6. Let us now examine the full key-recovery attack, using a short message M_1 of length 2^s. We run Step 2 until we gather $2^{2\ell/5}$ collisions, which requires about $2^{7\ell/10+1/2}$ calls to the MAC on average. Similarly, we run Step 3 until we recover the key; this should require $2^{3\ell/5}$ collisions on average, *i.e.* $2^{4\ell/5+1/2}$ evaluations of the compression function.

Since Step 0 still dominates, we use $s = 54$ to minimize the complexity:

Step 0:	$2^{415.1}$	**Step 1:**	$\ell \cdot 2^{\ell/2} = 2^{247}$
Step 2:	$2^{s+7\ell/10+1/2} = 2^{412.9}$	**Step 3:**	$2^{4\ell/5+1/2} = 2^{410.1}$

Therefore, the expected complexity is less than 2^{416}.

Finally, recall from Section 2 that our attacks use a modified scheme $\mathrm{MAC}_2(K, M) = \mathrm{MAC}_1(K, M \parallel [0]) \parallel \mathrm{MAC}_1(K, M \parallel [1])$, which results in a multiplicative penalty of at most 2. Hence, the total expected complexity is less than 2^{417}.

C Pseudo-code

Algorithm 1 Binary search matching

```
function SEARCH(x, d)                                  ▷ The message is a global parameter
    if |NEXT(x) = 0| then
        return NULL
    end if
    z ← x
    i ← d
    while i > 0 do
        if |NEXT(z) = 0| then
            return CUT(z)
        else if |NEXT(z) = 1| then
            z ← NEXT(z)
        else if |NEXT(z) = 2| then
            z_1, z_2 ← NEXT(z)
            if SIZE(z_1) > SIZE(x)/2 then
                z ← z_1
            else if SIZE(z_2) > SIZE(x)/2 then
                z ← z_2
            else
                if TEST (z_1) then                     ▷ Compare z_1 to a message node using a filter
                    return SEARCH(z_1, i)              ▷ We have SIZE(z_1) ≤ SIZE(x)/2
                else if TEST (z_2) then                ▷ Compare z_2 to a message node using a filter
                    return SEARCH(z_2, i)              ▷ We have SIZE(z_2) ≤ SIZE(x)/2
                else
                    return CUT(z)
                end if
            end if
        else if |NEXT(z) = 3| then
            ...                                        ▷ Handling a t > 2-collision is similar
        end if
        i ← i − 1
    end while
    if TEST (z) then
        return z
    else
        return CUT(z)
    end if
end function

function CUT(z)
    while |NEXT(PREV(z))| = 1 do
        z ← NEXT(PREV(z))
    end while
    Remove z subtree                                   ▷ We have SIZE(z) ≥ SIZE(x)/2
    return SEARCH(x, d)
end function
```