# Slide Attacks on a Class of Hash Functions

Michael Gorski[1], Stefan Lucks[1], and Thomas Peyrin[2]

[1] Bauhaus-University of Weimar, Germany {Michael.Gorski, Stefan.Lucks}@uni-weimar.de
[2] Orange Labs and University of Versailles Thomas.Peyrin@gmail.com

**Abstract.** This paper studies the application of slide attacks to hash functions. Slide attacks have mostly been used for block cipher cryptanalysis. But, as shown in the current paper, they also form a potential threat for hash functions, namely for sponge-function like structures. As it turns out, certain constructions for hash-function-based MACs can be vulnerable to forgery and even to key recovery attacks. In other cases, we can at least distinguish a given hash function from a random oracle.
To illustrate our results, we describe attacks against the Grindahl-256 and Grindahl-512 hash functions. To the best of our knowledge, this is the first cryptanalytic result on Grindahl-512. Furthermore, we point out a slide-based distinguisher attack on a slightly modified version of RadioGatún. We finally discuss simple countermeasures as a defense against slide attacks.

**Key words:** slide attacks, hash function, Grindahl, RadioGatún, MAC, sponge function.

## 1 Introduction

A hash function $H : \{0,1\}^* \rightarrow \{0,1\}^n$ is used to compute an $n$-bit fingerprint from an arbitrarily-sized input. Established security requirements for cryptographic hash functions are collision resistance, preimage and 2nd preimage resistance – but ideally, cryptographers expect a good hash function to somehow *behave like a random oracle*.

Current practical hash functions , such as SHA-1 or SHA-2 [25, 26], are *iterated* hash functions, using a *compression function* with a fixed-length input, say $h : \{0,1\}^{n+l} \rightarrow \{0,1\}^n$, and the Merkle-Damgård (MD) transformation [14, 24] for the full hash function $H$ with arbitrary input sizes. The core idea is to split the message $M$ into $l$-bit blocks $M_1, \ldots, M_m \in \{0,1\}^l$ (with some padding, to ensure all the blocks are of size $l$-bit), to define an *initial value* $X_0$, and to apply the recurrence $X_i = h(X_{i-1}, M_i)$. The final *chaining variable* $X_i$ is used as the hash output. The main benefit of the MD transformation is that it preserves collision resistance: if the compression function is collision resistant, then so is the hash function. Recent results, however, highlight some intrinsic limitations of the MD approach. This includes being vulnerable to multicollision attacks [16], long second-preimages attacks [19], and herding [18]. Even though the practical relevance of these attacks is unclear, they highlight some security issues, which designers of new hash functions should avoid.

In general, and due to certain structural weaknesses, MD-based hash functions do not *behave like a random oracles*. Consider, e.g., a secret key $K$, a message $M$ and define a Message Authentication Code $\mathrm{MAC}(K, M) = H(K||M)$. If we model $H$ as a random oracle, this construction meets the expected security requirements for a MAC. But for an MD-based hash function $H$, one can easily forge authentication codes: given $\mathrm{MAC}(K, M) = H(K||M)$, compute a valid $\mathrm{MAC}(K, M||Y) = H(K||M||Y)$ without knowing the secret key $K$. Coron et al. [11] recently discussed a formal model to prove hash functions being free from such structural weaknesses (but still weak against multicollision attacks).

**Our contribution.** Newly proposed hash function designs should not suffer from length extension. So for a new and well-designed hash function, the $\text{MAC}(K, M) = H(K||M)$ *should* be a secure MAC. We will show that this is not the case for some recently proposed hash functions. In contrast to the case of MD-based hash functions, where one can forge messages but cannot recover $K$, our attacks allow, in general, the adversary to find $K$ (much faster than by exhaustively searching for it).

Our attacks are an application of *slide attacks*. These are a classical tool for block ciphers cryptanalysis, but have so far not been used for hash function cryptanalysis.

**The Targets for Our Attacks.** A natural idea for thwarting the MD limitations is to increase the size of the internal chaining variables in the iterated process, see, e.g., [23]. Using a similar patch, sponge functions [3] followed the idea to employ a huge internal state (to hold a huge chaining variable) and to claim a *capacity* $c$, typically $c \gg n$. This defends against attackers even if these can perform $\gg 2^{n/2}$ operations (but are still restricted to $\ll 2^{c/2}$ units of work). Here $n$ is considered a typical hash function output size (sponge functions may also provide for arbitrary output sizes, rather than for a fixed $n$).

Several recent hash functions follow this approach, including GRINDAHL [22] and RADIOGATÚN [2]. As far as we know, there are no cryptanalytic attack on either RADIOGATÚN or the 512-bit version of GRINDAHL while some collision attacks for the 256-bit version of GRINDAHL have already been published [27, 20].

In the current paper, we study the applicability of slide attacks for sponge functions. Our results indicate that slide attacks can be a serious threat for hash functions fitting into the sponge framework. On the other hand, if the hash function designer is aware of slide attacks, we believe it is easy to defend against such attacks. We give concrete examples by providing attacks against GRINDAHL [22] and two slightly tweaked versions of RADIOGATÚN [2]. Our attack applies for both published flavours of Grindahl, the 256-bit version and the 512-bit version. As far as we know, this is the first cryptanalytic result for the 512-bit version.

**Outline:** in Section 2 we recall the slide attacks basics, study the case of hash functions and focus on the case of sponge functions. Then, in Section 3 we give an example by applying our results to the GRINDAHL hash function and discuss the vulnerability of RADIOGATÚN to slide attacks in Section 4. Finally, we describe cheap and simple defenses against slide attacks and conclude in Section 5.

## 2 Slide Attacks

Block ciphers are often designed as a keyed permutation which is applied many rounds. It is a common belief that increasing the number of rounds makes the cipher stronger, but this is just true for statistical attacks such as differential or linear cryptanalysis. Some attacks can be applied even for block cipher variants with an arbitrary number of rounds. This is true for certain related key attacks, and for slide attacks. The usual defense is to strengthen the key schedule and the keyed permutation itself. Related key attacks have been introduced by Biham [5] and independently by Knudsen [21]. Slide attacks [8] utilize the self-similarity of the cipher, typically caused by a periodic key schedule. An $r$ round block cipher with the same keyed permutation $F^i$ in each round can be attacked by slide attacks if $F^i$ is a weak permutation, i.e. the key used in $F^i$ can be found with a slid plaintext-ciphertext pair.

## 2.1 Slide Attacks on Block Ciphers

Slide attacks on block ciphers have been applied to some ciphers with a weak key schedule (see [8, 17, 12, 9, 28, 6, 29, 15]). The original slide attack [8] works as follows. An $n$-bit block cipher $E$ with $r$ rounds is split into $b$ identical rounds of the same keyed permutation $F^i$ for $i = \{1, \ldots, b\}$. In the simplest case we have $b = r$ where the key schedule produces the same key in each round[3]. Thus we write the cipher as $E = F^1 \circ F^2 \circ \cdots \circ F^b = F \circ F \circ \cdots \circ F$. A plaintext $P_j$ is then encrypted as

$$P_j \xrightarrow{F} X^{(1)} \xrightarrow{F} X^{(2)} \xrightarrow{F} \cdots \xrightarrow{F} C_j$$

where $X^{(i)}$ represents the intermediate encryption value after application of $F^i$ and $X^{(b)} = C_j$ is the corresponding ciphertext. To mount a slide attack one has to find a slid pair of plaintexts $(P_j, P_k)$, such that

$$P_k = F(P_j) \quad \text{and} \quad C_k = F(C_j) \tag{1}$$

hold, see also Figure 1.

$$\begin{array}{llllllllll}
P_j \xrightarrow{F} X^{(1)} \xrightarrow{F} X^{(2)} \xrightarrow{F} X^{(3)} \xrightarrow{F} \cdots \xrightarrow{F} C_j \\
\qquad\quad P_k \xrightarrow{F} X^{(1)} \xrightarrow{F} X^{(2)} \xrightarrow{F} \cdots \xrightarrow{F} X^{(b-1)} \xrightarrow{F} C_k
\end{array}$$

**Fig. 1.** A slide attack on block ciphers

Slide attacks can only be applied to a small class of ciphers with weak permutations periodic key schedules. A permutation is weak if, given the two equations in (1), it is easy to extract a non negligible part of the secret key. With $2^{n/2}$ known plaintext/ciphertext pairs $(P_i, C_i)$ we expect at least one pair satisfying $P_k = F^i(P_j)$ among these texts by the birthday paradox. This gives us a slid pair. Thus, the classical slide attack allows to recover the unknown key of an $n$-bit block cipher using $O(2^{n/2})$ known plaintexts.

Advanced sliding techniques like *complementation slide* and *sliding with a twist* were introduced by Biryukov and Wagner [9]. These techniques allow to attack ciphers with a more complex key schedule. The basic concept of complementation slide is to slide two encryptions against each other where the inputs to the rounds may have a difference which is canceled out by a difference in the keys, while an encryption is slid against a decryption using a sliding with a twist technique. The *realigning slide attack* [28] allows to slide encryptions with unslid rounds in the middle of the slide. Biham et al. [6] improved the slide attack to detect a large amount of slid pairs very efficiently by using the relation between the cycle structure of the entire cipher and that of the keyed permutation.

## 2.2 Slide Attacks on Hash Functions

Slide attacks in a hash function setting have attracted very few consideration in the literature. To our knowledge, the only paper considers an attack on the internal block cipher from SHA-1 [31].

[3] Note that $F^i$ might include more than one rounds of the cipher. If the key schedule produces identical keys with period $p$ then $F^i$ includes $p$ rounds of the original cipher.

However, yet no direct way to transform it into a practical attack on the hash function has been found.

Slide attacks for block ciphers are different in some aspects from those applied on hash functions. By definition, block cipher computations depend on a secret key, and slide attacks are typically employed to distinguish a block cipher from a random permutation – and often for a key recovery attack to follow.

In the hash function case, there is no secret key to recover, just the message to be hashed, and the adversary is allowed to know this message – or even to choose it. Typical attacks on hash functions are about finding collisions or preimages – and it is hard to see how slide attacks could be employed in that context. But even for hash functions, a slide property that (or which) can be detected with some significant probability will allow us to differentiate the scheme from a random oracle. Indeed, with such a property, one can show a non random behavior of the hash function. This is already an issue, since hash functions are often utilized to simulate a random oracle as they are considered to be the closest practical primitive to this theoretical object. Going further, when secret data is used as a part of the input of the hash function, one can try to recover some information from it. The natural primitive where hash functions handle secret data are of course the Message Authentication Codes (MAC), that permit to authenticate a message $M$ with a symmetric secret key $K$. For example, constructions such as HMAC [1] are implemented in a lot of different applications and make only two calls to a hash function. HMAC has the advantage to only require the internal function to be *weakly collision resistant* and also to provide secure MACs with MD-based hash functions. Note that a HMAC-based patch is one of the new domain extension algorithm proposed by Coron et al. [11] to thwart the simple MD-based MACs attacks. Those attacks are no more than a slide attack on the MD domain extension algorithm.

Generally, a good hash function $H$ should provide a good MAC with the following computations: $\mathrm{MAC}(K, M) = H(K||M)$ or $\mathrm{MAC}(K, M) = H(K||M||K)$. Just like for block ciphers, if the hash function considered is not protected, one may be able to recover some non negligible part of the secret key $K$ with a slide property that can be detected with a good probability. One has to note a work from Sasaki et al. [32] that attacks prefix, suffix and hybrid approaches for MAC constructions by using inner collisions for MD4, and a work from Preneel and Van Oorschot [30] that studies the envelope approache instantiated with MD5.

### 2.3 Slide attacks on "extended" sponge constructions

We analyze in this section how one can apply slide attacks to sponge-based hash functions, a newly introduced framework for building hash functions [2, 3]. More precisely, we use the "extended" sponge functions, a more general framework.

**The "extended" sponge framework** Assume that $H$ is an iterative hash function with an internal state of $c$ words of $p$-bit each and a final output size of $n$ bits. Let $M = M^1||M^2|| \cdots ||M^l$ be the $m \times p$-bit blocks of the message to hash with $M^l \neq 0^{m \times p}$ (the message is padded before split into blocks). Let $M^i$ be the message block hashed at each round $i$ and $X^i$ the internal state after proceeding $M^i$, with $X^0 = IV$. We then have $X^i = F(S(X^{i-1}, M^i))$, where $F$ is the round function and $S$ defines how the message is incorporated in the internal state. Once all the $l$ message blocks have been processed, $r$ blank rounds (rounds with no message input) are applied $X^i = F(X^{i-1})$ and $A := X^{l+r}$ is the final internal state. Finally, we derivate $n$ output bits by using the final output function $T(X^{l+r})$. Such a hash function can be written as

$$H(M) = X^0 \xrightarrow{F(S(X^0,M^1))} \cdots \xrightarrow{F(S(X^{l-1},M^l))} X^l \overbrace{\xrightarrow{F(X^l)} \cdots \xrightarrow{F(X^{l+r-1})}}^{\text{r times}} \overbrace{X_i^{l+r}}^{A} \xrightarrow{T(A)} T_A,$$

where $T_A$ represents the hash output. One has to note that for efficiency reasons and since the internal state will be big in practice, $F$ is usually a quite light and fast round permutation.

This framework is really general and especially more general than the original sponge function one. More precisely, in the original model, $S$ introduces the message blocks by XORing them to particular positions of the internal state. However, in our situation, we can also consider a function $S$ that replaces some bits of the internal state by the message bits. We call the former *XOR sponge* and the latter *overwrite sponge*. Moreover, in the original model, the final output function $T$ continues to apply some blank rounds and extract some bits from the internal state at the end of each application, until $n$ bits have been received. In our framework we can also consider the case where the output bits come from a direct truncation of the final internal state $A$, and we call it *truncated sponge*.

There are two security issues, related to the general design of sponge functions. One issue is *invertibility*: one can run the function $F$ into both directions. The second issue is *self-similarity*: all the blank rounds behave identically, and even a normal round can behave as a blank round if we have $X^{i-1} = S(X^{i-1}, M^i)$ (the effect of adding the message block is void). In the case of a XOR sponge we need $M^i = 0$ and in the case of an overwrite sponge we require that $M_i$ is equal to the overwritten part of the internal state.

We will exploit self-similarity for our slide attacks. The idea is that if one message $M_1 = M^1||\ldots||M^l$ is the prefix of another message $M_2 = M^1||\ldots||M^l||M^{l+1}$, the extended state after processing the first $l$ blocks is the same. Now, if $X^{l+1} = S(X^l, M^{l+1})$, processing the next message block $M^{l+1}$ for the longer message is the same as the first blank round when hashing the shorter message – the extended states remain identical. We call these two messages a *slid* pair : the two final internal states are just one permutation away $B := X_j^{l+r+1} = F(X_i^{l+r})$. The slide attack is shown in Figure 2.

$$\begin{aligned}
H(M_i) &= X_i^0 \xrightarrow{F(S(X_i^0,M^0))} \cdots \xrightarrow{F(S(X_i^{l-1},M^l))} X_i^l \overbrace{\xrightarrow{F(X_i^l)} \cdots\cdots\cdots\cdots \xrightarrow{F(X_i^{l+r-1})}}^{r\ \text{times}} \overbrace{X_i^{l+r}}^{A} \xrightarrow{T(A)} T_A \\
H(M_j) &= X_j^0 \xrightarrow{F(S((X_j^0,M^0))} \cdots \xrightarrow{F(S((X_j^{l-1},M^l))} X_j^l \xrightarrow{F(S((X_j^l,M^{l+1}))} X_j^{l+1} \underbrace{\xrightarrow{F(X_j^{l+1})} \cdots\cdots \xrightarrow{F(X_j^{l+r})}}_{r\ \text{times}} \underbrace{X_j^{l+r+1}}_{B} \xrightarrow{T(B)} T_B
\end{aligned}$$

**Fig. 2.** A slide attack on hash functions

Once we were able to generate a slid pair, we need to detect it. This fully depends on the output function $T$. When $T$ is defined as in the original sponge framework, it is very easy to detect a slid pair : most of the output bits will be equal, just shifted by one round. If $T$ is a truncation, we need to do a case by case analysis depending on the strength of the round function $F$ and the number of bits thrown away. Yet finding and detecting a slid pair already allows us to differentiate the hash function from a random oracle.

We can try to go further, by attacking a MAC with prefix key, i.e. MAC$(K, M)$. Note that such a construction makes sense as using HMAC based on a sponge hash function will turn out to be very inefficient. This is due to the fact that hashing very short messages (required in HMAC by the second hash function call) is quite slow because of the blank rounds. Therefore, Bertoni et al. [4] proposed to use prefix-MAC instead of HMAC.

Consider a secret key $K$. For simplicity and without loss of generality, we assume some $K$ to be a uniformly distributed $(k \times m \times p)$-bit random value (i.e. $k$ message words long), for some public integer constant $k$. We will write $K = (K^1, \ldots, K^m) \in (\{0,1\}^{m\times p})^k$. The adversary is allowed to choose message challenges $C_i$, while the oracle replies $\mathrm{MAC}(K, C_i) = H(K||C_i)$. Ideally, finding $K$ in such a scenario would require the adversary to exhaustively search over the set of all possible $K \in \{0,1\}^{k\times m\times p}$, thus taking $2^{k\times m\times p-1}$ units of time on average. Forging a valid MAC depends on the size of the hash output and the size of the key, with a generic attack it requires $\min\{2^{k\times m\times p-1}, 2^n\}$ units of time. A pair of challenges $(C_i, C_j)$, with $C_i = C_i^1||C_i^2||\cdots||C_i^l$ and $C_j = C_i||C_j^l$ is called a slid pair for $K$ if their final internal state are slid by one application of the blank round function as:

$$X_j^{k+l+r+1} = F(X_i^{k+l+r})$$

Provided that one can generate slid pairs and detect them, one can also try to retrieve the internal state $X_i^{k+l+r}$ thanks to this information. Again, a case by case analysis is required here. When $X_i^{k+l+r}$ is known, one can invert all the blank rounds and get $X_i^{k+l}$. Note that with this information, an attacker can directly forge valid MACs for any message that contains $M$ as prefix (exactly like the extension attacks against MD-based hash functions). If the round function with the message is also invertible, we can continue to invert all the challenge rounds and get $X_i^k$. This will allow us to recover some non trivial information on the secret key $K$.

A general outline of the attack is as follows:

1. Find and detect slid pairs of messages
2. Recover the internal state
3. Uncover some part of the secret key or forge valid MACs

The padding is very important here : for the XOR sponge functions, an appropriate padding can avoid slide attacks. Indeed, in that case, we require $M^l = 0^{m\times p}$ to get a slid pair. This gives an explanation why the condition $M^l \neq 0^{m\times p}$ is needed for the indifferentiability proofs of XOR sponge functions. However, for the truncated sponge function, a padding is ineffective to avoid slide attacks.

## 3 Applications

### 3.1 The Grindahl Design

Grindahl is a new hash function introduced by Knudsen et al. in [22], that fits our extended sponge framework. More precisely, it is an overwrite sponge function. There are two concrete instantiations of the Grindahl hash function family: a 256-bit and a 512-bit hash function proposed in the original Grindahl paper [22]. The parameters of these instantiations in our framework are defined as follows:

**Grindahl-256 [22].** Grindahl-256 is a 256-bit hash function with $N_r = 4$ and $N_c = 12$. The rotation amounts are $(\rho_0, \ldots, \rho_3) = (1, 2, 4, 10)$.

**Grindahl-512 [22].** Grindahl-512 is a 512-bit hash function with $N_r = 8$ and $N_c = 12$. The rotation amounts are $(\rho_0, \ldots, \rho_7) = (1, 2, \ldots, 8)$.

Note that the internal state of GRINDAHL can also be viewed as a matrix. Therefore, we define $N_r$ and $N_c$ to be the number of rows and columns of $p$-bit word respectively: we have $N_r \times N_c = c$. For each instance of GRINDAHL we have $p = 8$. The message chunk entering at each round can then be viewed as one column, thus $m = N_r$.

For GRINDAHL the padding consists of 10- and length-padding:

**1.** *10-padding* appends a "1"-bit to the message, followed by as many "0"-bits as needed to complete the last message block.

**2.** *Length-padding* then appends the number of message blocks (not bits!) for the entire padded message as a 64-bit value.

One effect of the 10-padding is that the last message block before the Length-padding can be any value, except for the all-zero block. (Or equivalently, any nonzero block $B$ can be split up into an incomplete block $R$ plus 10-padding: $B = R + P^{\text{“}10''}$. Note that $R$ is 0 bit long if $B = 1000\ldots0$.)

A message $M = M^1||\ldots||M^l$ of 32-bit blocks $M^i$ in the case of GRINDAHL-256, and an incomplete block $M^l$, will be padded to $Pad(M) = M^1||\ldots||M^l + P_1^{\text{“}10''}||M^{l+1}||M^{l+2}$, where $P_1^{\text{“}10''}$ is the 10-padding. This padded message has the following properties:

1. The last-but-two message block is not zero: $M^l + P_1^{\text{“}10''} \neq 0^{32}$.
2. The final two message blocks contain the 64-bit integer $l$: $(M^{l+1}||M^{l+2}) = l$. (From the GRINDAHL sample implementation, we conclude that the 32 least significant bits of the 64-bit value are stored in $M^{l+2}$, while the high-significant bits go into $M^{l+1}$.)

Similarly for GRINDAHL-512, a message $M = M^1||\ldots||M^l$ of 64-bit blocks $M^i$, where $M^l$ is also incomplete, is padded to $Pad(M) = M^1||\ldots||M^l + P_1^{\text{“}10''}||M^{l+1}$ has the following properties after padding:

1. The last-but-one message block is not zero: $M^l + P_1^{\text{“}10''} \neq 0^{64}$.
2. The last message block contains the 64-bit integer $l$: $M^{l+1} = l$.

Most hash functions for variably-sized inputs iterate an underlying compression function for fixed-size inputs. GRINDAHL is no exception. At the end, the output will be the first $n/(p \times N_r)$ columns of of the final internal state. I.e., GRINDAHL is a truncated sponge. Internally, GRINDAHL uses a state of $(N_r \times N_c)$ words of $p$ bit each. The compression function takes one $m$-word message block and an $(N_r \times N_c)$-word internal state as its input and generates new internal state (again of the size $(N_r \times N_c)$ words, of course), as its output.

Regarding this compression function, GRINDAHL follows a general three-step design strategy. Assume a $m$-word message block, which we write as $M^i$ and a $(N_r \times N_c)$-word internal state, which we write as a $N_c$ tuple of $N_r$-words: $(X^1, \ldots, X^{N_c}) \in (\{0,1\}^{p \times N_r})^{N_c}$. The incorporation step which concatenates a message block to the internal state is straightforward:

$$\text{S}: \{0,1\}^{p \times N_r} \times \{0,1\}^{p \times N_r \times N_c} \to \{0,1\}^{p \times N_r + p \times N_r \times N_c}, \text{S}(M^i, (X^1, \ldots, X^{N_c}) = (M^i, X^1, \ldots, X^{N_c}).$$

The $(p \times N_r + p \times N_r \times N_c)$-bit output of the incorporating $S$ is the *extended state* $(X^0, \ldots, X^{N_c})$. The second step is a permutation over the extended state:

$$\text{F: } \{0,1\}^{p\times N_r + p\times N_r \times N_c} \to \{0,1\}^{p\times N_r + p\times N_r \times N_c},\ \text{F}(X^0, \ldots, X^{N_c}) = (Y^0, \ldots, Y^{N_c}).$$

$F$ is a permutation based on RIJNDAEL [13] primitives:

$$\text{F}(X^0, \ldots, X^{N_c}) = \text{MixColumns} \circ \text{ShiftRows} \circ \text{SubBytes} \circ \text{AddConstant}(X^0, \ldots, X^{N_c}).$$

**MixColumns.** Is a linear matrix multiplication of each state matrix column with a constant vector. This transformation is defined as in the RIJNDAEL specifications for the 256-bit version of GRINDAHL.

**ShiftRows.** This transformation cyclically shifts bytes a number of positions along each row. Thus, the $i$-th row is rotated by $\rho_i$ positions to the right.

**SubBytes.** The only non-linear part of the permutation, exactly defined as the SubBytes function of RIJNDAEL.

**AddConstant.** This function is a simple XORing of the state matrix with a constant matrix $M$ of the same size, where all bytes are zero except for one.

See [22] for a detailed description of GRINDAHL. The third operation is as straightforward as the first one – the first $p \times N_r$-bits of the $(p \times N_r + p \times N_r \times N_c)$-bit extended state are truncated away, to get a new $p \times N_r \times N_c$-bit internal state $(Y^1, \ldots, Y^{N_c})$:

$$\text{R: } \{0,1\}^{p\times N_r + p\times N_r \times N_c} \to \{0,1\}^{p\times N_r \times N_c},\ \text{R}(Y^0, \ldots, Y^{N_c}) = (Y^1, \ldots, Y^{N_c}).$$

See Figure 3 for a visual illustration of this design strategy. Note that the final truncation in one iteration and the initial concatenation of the $b$-bit message block in the next iteration together are tantamount to simply overwriting the corresponding column of the extended internal state. The final truncation is specified as

$$\text{T: } \{0,1\}^{p\times N_r + p\times N_r \times N_c} \to \{0,1\}^{n},\ \text{T}(Y^0, \ldots, Y^{N_c}) = (Y^1, \ldots, Y^{n/(p\times N_r)}).$$

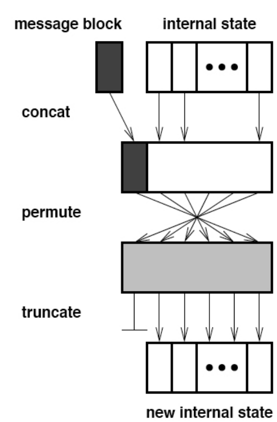


**Fig. 3.** The general design of the GRINDAHL compression function.

Let $\alpha$ be the internal state matrix with $N_c$ columns and $N_r$ rows, while $\hat{\alpha}$ represents the extended internal state with $N_c + 1$ columns and $N_r$ rows. For a padded message $M = M^1 || \ldots || M^d$ the

GRINDAHL hash function does for $0 < i < d$:

$$\alpha \leftarrow R(P(S(M^i, \alpha)))$$

For the last message input $M^d$ GRINDAHL performs $\hat{\alpha} \leftarrow P(S(M^d, \alpha))$. The truncation $R$ is omitted after the last message input and finally 8 blank rounds with no message input are performed. These rounds only consists of the $P$ operation on $\hat{\alpha}$. The final output remains after performing the output truncation $T$, which leaves the $n$-bit output.

### 3.2 Slide attacks on GRINDAHL-512

**Find slid pairs of messages** Building the challenge that generates a slid pair works as follows. We choose a message $M_1 = M_1^0||M_1^1||\dots||M_1^{l-1}||M_1^l$, where $M_1^l$ is a non complete block which will be padded. The MAC therefore processes

$$Pad(K||M_1) = K||M_1^0||M_1^1||\dots||M_1^{l-1}||M_1^l + P_1^{\text{“10”}}||P_1^L$$

where $P_1^{\text{“10”}}$ is the 10-padding to $M_1^l$ and $P_1^L$ is the one-block of the message length. The value of $P_1^L$ can be chosen by the attacker while modifying the message length. For each $M_1$ we build the message $M_2 = M_1^0||M_1^1||\dots||M_1^{l-1}||M_1^l + P_1^{\text{“10”}}||R$, where $R$ is a random incomplete block. The MAC proceeds

$$Pad(K||M_2) = K||M_1^0||M_1^1||\dots||M_1^{l-1}||M_1^l + P_1^{\text{“10”}}||R + P_2^{\text{“10”}}||P_2^L$$

and in some cases we have

$$Pad(K||M_2) = K||M_1^0||M_1^1||\dots||M_1^{l-1}||M_1^l + P_1^{\text{“10”}}||P_1^L||P_2^L.$$

The messages $M_1$ and $M_2$ only differ in one additional block at the end. A pair $(M_1, M_2)$ will be a slid pair with probability $2^{-64}$. Detecting a slid pair is quite simple. Let $T_A = A^0, \dots, A^7$ and $T_B = B^0, \dots, B^7$ be the query output (the truncated final internal states $A$ and $B$). Then the condition $B = P(A)$ holds for a slid pair only. We could not directly apply another blank round to $A$ since we only know $T_A$ and not $A$. However, $T_A$ and $T_B$ leave enough information for detecting a slid pair. We can invert $T_B$ one blank round and compare the resulting bytes with the bytes known from $T_A$. Thus, we can compare 34 bytes of $T_A$ with the known bytes obtained from inverting $T_B$. In this way we can detect a slide pair since one occurs among $2^{64}$ pairs. Using the computation described above we can filter $2^{8 \cdot 34} = 2^{272}$ false pairs. Figure 4 shows the backward computation of one blank round.

**Recover the internal state** A challenge $(M_1, M_2)$ which produces a slid pair $(T_A, T_B)$ can be used to recover the final internal state $A$ (corresponding to the computation of $M_1$) just before the final truncation. Since the columns $A^8$ to $A^{12}$ are unknown, we have to recover 40 bytes. As shown in Figure 4, we can directly recover 30 bytes from $A$ by computing $T_B$ one blank round backward, exactly as when we tried to detect slid pairs: we can fully invert the MixColumns transformation for the eight first columns (where all the bytes are known), then it is also very easy to invert ShiftRows, SubBytes and AddConstant transformations. So, when looking at Figure 4, it is clear that the attacker can directly get 30 unknown bytes from $A$. The remaining 10 unknown bytes can

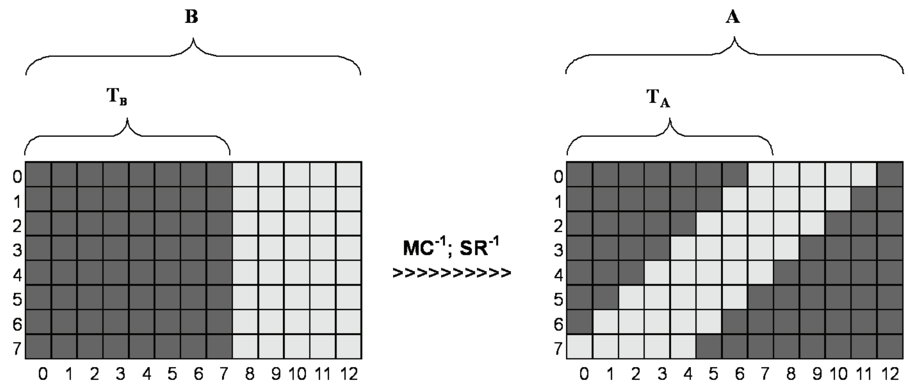


**Fig. 4.** Detecting a slid pair of messages for GRINDAHL-512. Cells in dark gray mark known bytes while cells in light gray mark unknown bytes. The inverse MixColumns ($MC^{-1}$) and the inverse ShiftRows ($SR^{-1}$) are the only two operations which are important for our analysis: AddConstant and SubBytes functions leave a known (respectively unknown) bytes known respectively unknown). Therefore we prevent the other operations.

be recovered in a different way. For each possibility among those bytes ($2^{8 \cdot 10} = 2^{80}$ possibilities), we invert all the blank rounds and check if the last added word (the first encountered when computing backward) is $P_1^L$. Indeed, when inverting the real internal state $A$, we surely come to the insertion of $P_1^L$ and this can be easily detected since we know this message block and since the message insertion overwrite the first column of the internal state. Now we are dealing with $2^{80-64} = 2^{16}$ possibilities only and we have to be careful, since some bytes become undetermined, if we continue the backward computation. The undetermined bytes are those which are replaced by the inserted message word during the message input step (due to the overwriting). However, we don't need them to discriminate among the $2^{16}$ lasting possibilities and we can compute one more round backward to check if we finally obtain the message word $M_1^l + P_1^{\text{“10”}}$ inserted. This leaves us the complete internal state $A$.

**Uncover some parts of the secret key or forge valid MACs** By knowing the whole internal state $A$ it is straightforward to invert the blank rounds. With this information, we can directly generate new valid MACs for messages which contain $M_1$ as prefix: we just have to continue the computation of the hash function by ourselves.

We can also try to invert the rounds where known message words are inserted. Some parts of the internal state are undetermined because of the truncation when adding message words as mentioned in the previous section. We can guess those undetermined columns by only keeping those which lead to the good inserted message words in the first column. This is equal to what we did above to recover the final internal state. By trying all the possible values of the truncated column, we can continue going backward and check which one leads to the known correct values of the message blocks inserted a few rounds before. Some trials will lead to wrong message blocks inserted and can be discarded. The one leading to the good values have a good chance to be the real erased bytes. Thus, we can go backward for all the known message words and recover the erased columns until we have to stop this procedure when we reach the unknown secret key word. The last unknown column which can be recovered is the column before inserting $M_1^2$. Now, with all those informations we can recover 4 bytes from 8 of the last unknown message block we encounter (the first when computing backward), which is part of the secret key. The rest of the secret can be then computed exhaustively (at a lower cost than brute force without slide attacks) or we can use a trick[4]. Indeed, we know that the initial internal state is equal to zero and one can accelerate the secret recovery with a meet-in-the-middle attack: we compute forward from the known initial internal state and we compute backward as we described before.

### 3.3 Slide attacks on Grindahl-256

Applying the slide attack on Grindahl-256 is a little bit more difficult than on the 512 bit version, since the message block size is of 32 bit an the padding adds two additional blocks to the message. This makes it harder to control the message words and to find a slid pair. We describe the slide attack on Grindahl-256 in Appendix A.

## 4 Slide attacks on modified versions of RadioGatún

We are able to use the presented technique to attack slightly modified versions of RadioGatún [2]. There are two possible modifications. Either we change the padding rule such that the last message block can also be an all zero input block. Or we change the message input step such that the input block enters the state via a replacement of the current state column. I.e., we turn RadioGatún from an XOR sponge into an overwrite sponge. This modification is inspired by the message input step of Grindahl.

Consider the first case. The padding rule requires the final message block always to be non-zero, e.g., by applying the usual 10-padding. For an application where the message length always happens to be a multiple of the block size, this padding may appear to be moot. So consider an implementation without padding. Now the final message block might be all-zero. This gives an easy way to generate slid pairs $(M_i, M_j)$ of messages – just take any $M_i$ and set $M_j := (M_i||0)$ ($M_i$, concatenated by an all-zero message block). In this case, slide attacks are straightforward. Given for example a MAC such as

$$H(K||M_i) = Z_i^1, Z_i^2, Z_i^3\ , \dots, Z_i^k \quad \text{and}$$
$$H(K||M_i||M^{zero}||M^{zero}) = Z_i^3\ , \dots, Z_i^k, Z_i^{k+1}, Z_i^{k+2},$$

where $Z_i^r$ represents the r-th output stream, one can easily forge the MAC $Z_i^2, \dots, Z_i^{k+1}$, for the message $M_i||M^{zero}$.

For the second case (turning RadioGatún into an overwrite sponge), consider a pair of messages $M_i = M_i^1|| \dots ||M_i^d$ and $M_j = M_i||M_j^{d+1}$, with $M_i$ being a prefix of $M_j$ and $M_j$ being one block longer. Both final blocks $M_i^d$ and $M_j^{d+1}$ being non-zero are slid with a probability of $2^{-p\times m}$. It is easy to detect slid pairs by comparing $k-1$ of the output blocks. If the pair $(M_i, M_j)$ is slid, then we obtain:

$$H(K||M_i) = Z_i^1, Z_i^2\ , Z_i^3, \dots, Z_i^k \quad \text{and}$$
$$H(K||M_i||M_j^{d+1}) = Z_i^2\ , Z_i^3, \dots, Z_i^k, Z_i^{k+1}$$

[4] If the size of the key is not too big, we don't even require to do any exhausive search.

This shows that our slide attack can be used to distinguish some hash functions, e.g. sponge-based one, from a random oracle if the designer do not take care to avoid sliding properties of their hash functions.

Slide-like distinguishing attacks are also applicable for other schemes, i.e. a modified version of PANAMA even leaves more non-trivial information of the internal state than our attack on modified RadioGatún.

## 5 Possible Countermeasures and Conclusion

It only takes a negligible effort to defend hash functions from against slide attacks. Hash function designers, like block cipher designers, must be aware of possible slide attacks and be on guard for too much self-similarity in their constructions. For sponge-based hash functions, a simple patch would be to just add a nonzero constant just before running the blank rounds and extracting the hash value. Another option would be to marginally change the blank rounds. E.g., Grindahl could be changed such that the blank rounds use different rotation amounts (while maintaining the old rotation amounts for all the other rounds). Well-chosen padding rules also help. In the case of xor sponges, a good padding even seems to suffice as a defense against slide attacks.

We have studied the applicability of slide attacks for sponge functions. These are a classical tool for block cipher cryptanalysis, but have not been used for hash function cryptanalysis so far. Our results indicate that slide attacks can be a serious threat for sponge-based hash functions. If the hash function designer is aware of slide attacks, we believe that it is easy to defend against slide attacks. In our slide attacks on Grindahl and modified version of RadioGatún we demonstrated the power of these attacks. Our attacks apply for both published flavours of Grindahl, the 256-bit version and the 512-bit version. As far as we know, this is the first cryptanalytic result for the 512-bit version.

## Acknowledgements

The authors wish to thank the anonymous reviewers for helpful comments.

## References

1. Mihir Bellare, Ran Canetti, and Hugo Krawczyk. Keying hash functions for message authentication, 1996.
2. Guido Bertoni, Joan Daemen, Michael Peeters, and Gilles Van Assche. Radiogatun, a belt-and-mill hash function. Presented at Second Cryptographic Hash Workshop, Santa Barbara (August 24-25, 2006). See http://radiogatun.noekeon.org/.
3. Guido Bertoni, Joan Daemen, Michael Peeters, and Gilles Van Assche. On the Indifferentiability of the Sponge Construction. In Nigel P. Smart, editor, *EUROCRYPT*, volume 4965 of *Lecture Notes in Computer Science*, pages 181–197. Springer, 2008.
4. Guido Bertoni, Joan Daemen, Michael Peeters, and Gilles Van Assche. Sponge Functions, presented at ECRYPT Hash Workshop 2007.
5. Eli Biham. New Types of Cryptanalytic Attacks Using Related Keys. *J. Cryptology*, 7(4):229–246, 1994.
6. Eli Biham, Orr Dunkelman, and Nathan Keller. Improved Slide Attacks. In Biryukov [7], pages 153–166.
7. Alex Biryukov, editor. *Fast Software Encryption, 14th International Workshop, FSE 2007, Luxembourg, Luxembourg, March 26-28, 2007, Revised Selected Papers*, volume 4593 of *Lecture Notes in Computer Science*. Springer, 2007.
8. Alex Biryukov and David Wagner. Slide Attacks. In Lars R. Knudsen, editor, *FSE*, volume 1636 of *Lecture Notes in Computer Science*, pages 245–259. Springer, 1999.

9. Alex Biryukov and David Wagner. Advanced Slide Attacks. In *EUROCRYPT*, pages 589–606, 2000.
10. Gilles Brassard, editor. *Advances in Cryptology - CRYPTO '89, 9th Annual International Cryptology Conference, Santa Barbara, California, USA, August 20-24, 1989, Proceedings*, volume 435 of *Lecture Notes in Computer Science*. Springer, 1990.
11. Jean-Sébastien Coron, Yevgeniy Dodis, Cécile Malinaud, and Prashant Puniya. Merkle-Damgård Revisited: How to Construct a Hash Function. In Victor Shoup, editor, *CRYPTO*, volume 3621 of *Lecture Notes in Computer Science*, pages 430–448. Springer, 2005.
12. Nicolas T. Courtois, Gregory V. Bard, and David Wagner. Algebraic and Slide Attacks on KeeLoq. Cryptology ePrint Archive, Report 2007/062, 2007. http://eprint.iacr.org/.
13. Joan Daemen and Vincent Rijmen. *The Design of Rijndael: AES - The Advanced Encryption Standard*. Springer, 2002.
14. Ivan Damgård. A Design Principle for Hash Functions. In Brassard [10], pages 416–427.
15. Soichi Furuya. Slide Attacks with a Known-Plaintext Cryptanalysis. In Kwangjo Kim, editor, *ICISC*, volume 2288 of *Lecture Notes in Computer Science*, pages 214–225. Springer, 2001.
16. Antoine Joux. Multicollisions in Iterated Hash Functions. Application to Cascaded Constructions. In Matthew K. Franklin, editor, *CRYPTO*, volume 3152 of *Lecture Notes in Computer Science*, pages 306–316. Springer, 2004.
17. Selçuk Kavut and Melek D. Yücel. Slide Attack on Spectr-H64. In Alfred Menezes and Palash Sarkar, editors, *INDOCRYPT*, volume 2551 of *Lecture Notes in Computer Science*, pages 34–47. Springer, 2002.
18. John Kelsey and Tadayoshi Kohno. Herding Hash Functions and the Nostradamus Attack. In Serge Vaudenay, editor, *EUROCRYPT*, volume 4004 of *Lecture Notes in Computer Science*, pages 183–200. Springer, 2006.
19. John Kelsey and Bruce Schneier. Second Preimages on n-Bit Hash Functions for Much Less than $2^{\mathrm{n}}$ Work. In Ronald Cramer, editor, *EUROCRYPT*, volume 3494 of *Lecture Notes in Computer Science*, pages 474–490. Springer, 2005.
20. Dmitry Khovratovich. Cryptanalysis of hash functions with structures, presented at ECRYPT Hash Workshop 2008.
21. Lars R. Knudsen. Cryptanalysis of LOKI91. In Jennifer Seberry and Yuliang Zheng, editors, *ASIACRYPT*, volume 718 of *Lecture Notes in Computer Science*, pages 196–208. Springer, 1992.
22. Lars R. Knudsen, Christian Rechberger, and Søren S. Thomsen. The Grindahl Hash Functions. In Biryukov [7], pages 39–57.
23. Stefan Lucks. A Failure-Friendly Design Principle for Hash Functions. In Bimal K. Roy, editor, *ASIACRYPT*, volume 3788 of *Lecture Notes in Computer Science*, pages 474–494. Springer, 2005.
24. Ralph C. Merkle. One Way Hash Functions and DES. In Brassard [10], pages 428–446.
25. National Institute of Standards and Technology. FIPS 180-1: Secure Hash Standard. April 1995. See http://csrc.nist.gov.
26. National Institute of Standards and Technology. FIPS 180-2: Secure Hash Standard. August 2002. See http://csrc.nist.gov.
27. Thomas Peyrin. Cryptanalysis of Grindahl. In *ASIACRYPT*, pages 551–567, 2007.
28. Raphael Chung-Wei Phan. Advanced Slide Attacks Revisited: Realigning Slide on DES. In Ed Dawson and Serge Vaudenay, editors, *Mycrypt*, volume 3715 of *Lecture Notes in Computer Science*, pages 263–276. Springer, 2005.
29. Raphael Chung-Wei Phan and Soichi Furuya. Sliding Properties of the DES Key Schedule and Potential Extensions to the Slide Attacks. In Pil Joong Lee and Chae Hoon Lim, editors, *ICISC*, volume 2587 of *Lecture Notes in Computer Science*, pages 138–148. Springer, 2002.
30. Bart Preneel and Paul C. van Oorschot. On the security of two mac algorithms. In *EUROCRYPT*, pages 19–32, 1996.
31. Markku-Juhani Olavi Saarinen. Cryptanalysis of Block Ciphers Based on SHA-1 and MD5. In Thomas Johansson, editor, *FSE*, volume 2887 of *Lecture Notes in Computer Science*, pages 36–44. Springer, 2003.
32. Yu Sasaki, Lei Wang, Kazuo Ohta, and Noboru Kunihiro. Password Recovery on Challenge and Response: Impossible Differential Attack on Hash Function. In *AFRICACRYPT*, 2008.

## A A slide attack on Grindahl-256

### A.1 Find slid pairs of messages

Building the challenge that generates a slid pair works as follows. We choose a message $M_1 = M_1^0||M_1^1||\ldots||M_1^{l-1}||M_1^l$, where $M_1^l$ is a non complete block which will be padded. The MAC therefore processes the hash input

$$Pad(K||M_1) = K||M_1^0||M_1^1||\dots||M_1^{l-1}||M_1^l + P_1^{\text{“10”}}||P_1^{L1}||P_1^{L2},$$

where $P_1^{\text{“10”}}$ is the 10-padding to $M_1^l$ and $P_1^{L1}||P_1^{L2}$ is the two-block of the message length. Before building the second message, we want the condition

$$0^n \neq P_1^{L1} = P_1^{L2}$$

to always hold for $M_1$. Then, for each $M_1$ we build the message $M_2 = M_1^0||M_1^1||\dots||M_1^{l-1}||M_1^l + P_1^{\text{“10”}}||R$, where $R$ is an incomplete block which, after 10-padding, is the same as $P_1^{L_1}$. As $P_1^{L_1}$ is nonzero, such an $R$ exists. In this case, the hash input is

$$\begin{aligned} Pad(K||M_2) &= K||M_1^0||M_1^1||\dots||M_1^{l-1}||M_1^l + P_1^{\text{“10”}}||\; R + P_2^{\text{“10”}} \;||P_2^{L1}||P_2^{L2} \\ &= K||M_1^0||M_1^1||\dots||M_1^{l-1}||M_1^l + P_1^{\text{“10”}}|| \quad P_1^{L1} \quad ||P_1^{L2}||P_2^{L2} \end{aligned}$$

This holds because of the conditions fulfilled by $P_1^{L1}$ and $P_1^{L2}$. In other words, $M_1$ and $M_2$ only differ in an additional block at the end. Such a pair $(M_1, M_2)$ is slid with a probability of $2^{-32}$. Detecting a slid pair is as simple as in the case of GRINDAHL-512. Here also the condition $B = P(A)$ holds for a slid pair only. $T_A$ leaves enough information to compute column $B^4$ by performing one blank round on $T_A$. In this way the output $(T_A, T_B)$ of a challenge $(M_1, M_2)$ can be checked for a value of $B^4$ what we will expect for a slid pair. We can further check by using other columns than $B^4$, even if for them only a subspace of the potential solutions are determined by $T_A$. On the average, we need $2^{31}$ pairs until we find a slid one. Thus, we need to make about $2^{32}$ function calls to obtain and detect a slid pair. Figure 5 shows the backward computation of one blank round.

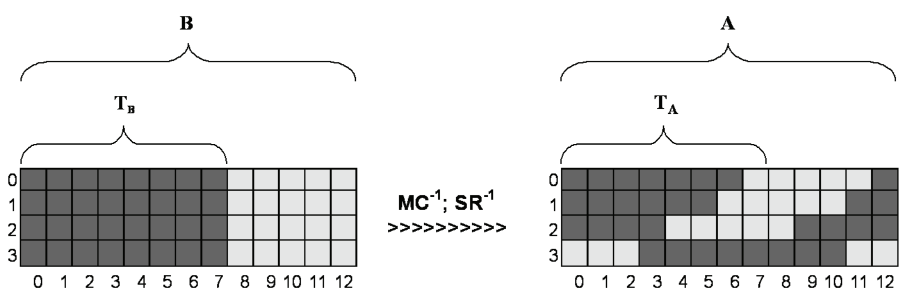


**Fig. 5.** Detecting a slide pair of messages for GRINDAHL-256. Cells in dark gray mark known bytes while cells in light gray mark unknown bytes. The inverse MixColumns ($MC^{-1}$) and the inverse ShiftRows ($SR^{-1}$) are the only two operations which are important for our analysis: AddConstant and SubBytes functions leave a known (respectively unknown) bytes known respectively unknown). Therefore we prevent the other operations.

### A.2 Recover the internal state

A challenge $(M_1, M_2)$ which produces a slid pair $(T_A, T_B)$ can be used to recover the final internal state $A$ (corresponding to the computation of $M_1$) just before the final truncation. Since the columns $A^8$ to $A^{12}$ are unknown we have to recover 20 bytes. We can directly recover 10 bytes from $A$ by computing $T_B$ one blank round backward, exactly as when we tried to detect slid pairs: we can fully invert the MixColumns transformation for the eight first columns (where all the bytes are known), then it is also very easy to invert ShiftRows, SubBytes and AddConstant transformations. So, when looking at Figure 5, it is clear than the attacker can directly get 10 unknown bytes from $A$. The remaining 10 unknown bytes can be recovered in a different way. For each possibility among those bytes ($2^{8 \cdot 10} = 2^{80}$ possibilities), we invert all the blank rounds and check if the last added word (the first encountered when computing backward) is $P_1^{L2}$. Indeed, when inverting the real internal state $A$, we surely come to the insertion of the block $P_1^{L2}$ and this can be easily detected since we know this message block and since the message insertion overwrite the first column of the internal state. We can continue to compute backward with the word $P_1^{L1}$ even if some parts of the internal state at this point becomes undetermined due to the truncation when inserting the message words and thus we only have $2^{48-32} = 2^{16}$ possibilities. Finally, we can continue to the message word $M_1^l + P_1^{\text{“10”}}$ which leads to a recovery of the full internal state $A$.

### A.3 Using only short messages

Note that the above attack required $0^n \neq P_1^{L1} = P_1^{L2}$, i.e., the most significant and the least significant word of the length field of $(K||M_1)$ must the same – and nonzero. Thus, the smallest possible choice for $P_1^{L1} = P_1^{L2}$ is $P_1^{L1} = P_1^{L2} = 1$, implying a message length (for $(K||M)$, i.e., including the key) of $1 + 2^{32}$ blocks. If dealing with such long messages is an issue, we can modify the attack so use short messages. The modified attack goes as follows.

We choose a message $M_1 = M_1^0||M_1^1||\dots||M_1^{l-1}||M_1^l + P_1^{\text{“10”}}$, where the final block $M_1^l$ is incomplete. The MAC processes the hash input

$$Pad(K||M_1) = K||M_1^0||M_1^1||\dots||M_1^{l-1}||M_1^l||P_1^{L1}||P_1^{L2},$$

with a length-field $P_1^{L1}||P_1^{L2}$. Note that $P_1^{L2}$ holds the 32 least significant bits, while $P_1^{L1}$ holds the 32 most significant bits. We assume short messages, thus $P_1^{L1} = 0^n$. This time, we want the MAC to process the hash input

$$\begin{aligned} Pad(K||M_2) &= K||M_1^0||M_1^1||\dots||M_1^{l-1}||M_1^l + P_1^{\text{“10''}}||P_1^{L1}||\ S + P_2^{\text{“10''}}\ ||P_2^{L1}||P_2^{L2} \\ &= K||M_1^0||M_1^1||\dots||M_1^{l-1}||M_1^l + P_1^{\text{“10''}}||P_1^{L1}|| \quad\ \ P_1^{L2} \quad\ \ ||P_2^{L1}||P_2^{L2}, \end{aligned}$$

Thus, $M_1$ and $M_2$ only differ in *two* additional blocks at the end. Accordingly, we choose

$$M_2 = M_1^0||M_1^1||\dots||M_1^{l-1}||M_1^l + P_1^{\text{“10''}}||P_1^{L1}||S.$$

As $P_1^{L2}$ is nonzero, an incomplete block $S$ with $S + P_2^{\text{“10''}} = P_1^{L2}$ does exist.

Now we define $M_1$ and $M_2$ as a slid-by-two pair, if, when processing the shorter message $M_1$, the first two empty rounds behave exactly the last two nonempty rounds when processing $M_2$. This happens with a probability of $(2^{-32})^2$, and on the average, we need $2^{63}$ pairs to find slid-by-two pair.

A pair of messages is slid-by-two, if and only if the two corresponding states $A$ and $B$ satisfy $B = P(P(A))$. Detecting slid-by-two pairs from $T(A)$ and $T(B)$ and then recovering the internal state $A$ is slightly more complicated, compared to “ordinarily” slid-by-one pairs, but still feasible.

### A.4 Uncover some parts of the secret key or forge valid MACs

By knowing the whole internal state $A$ it is straightforward to invert the blank rounds. With this information, we can directly generate new valid MACs for messages which contain $M_1$ as prefix: we just have to continue the computation of the hash function by ourselves.

We can also try to invert the rounds where known message words are inserted. Some parts of the internal state are undetermined because of the truncation when adding message words. We do not known what was in the first column before erasing it with a message word, except for the first undetermined column which is equal to $P_1^{L2}$ as described above. But we can guess those undetermined columns by only keeping those which lead to the good inserted message words in the first column. This is equal to what we did above to recover the final internal state. By trying all the possibles values the truncated column, we can continue going backward and check which one leads to the known correct values of the message blocks inserted a few rounds before. Some tries will lead to wrong message blocks inserted and can be discarded. The one leading to the good values have a good chance to be the real erased bytes. Thus, we can go backward for all the known message words and recover the erased columns until we have to stop this procedure when we reach the unknown secret key word. The last unknown column which can be recovered is the column before inserting $M_1^3$. Now, with all those informations we can recover 1 bytes from 4 of the last unknown message block we encounter (the first when computing backward), which is part of the secret key. The rest of the secret can be then computed exhaustively (at a lower cost than brute force without slide attacks) or we can use a trick[5]. Indeed, we know that the initial internal state is equal to zero and one can accelerate the secret recovery with a meet-in-the-middle attack: we compute forward from the known initial internal state and we compute backward as we described before.

[5] If the size of the key is not too big, we don't even require to do any exhausive search.