Speeding Up The Widepipe: Secure and Fast Hashing

Mridul Nandi[†] and Souradyuti Paul[†‡]

[†]National Institute of Standards and Technology
Security Technology Group
100 Bureau Dr., MS 8931
Gaithersburg, MD 20899, United States.
[‡]Katholieke Universiteit Leuven, Dept. ESAT/COSIC,
Kasteelpark Arenberg 10,
B–3001, Leuven-Heverlee, Belgium.
`mridul.nandi@gmail.com, souradyuti.paul@nist.gov`

Abstract

In this paper we propose a new sequential mode of operation – called the *Fast wide pipe* or FWP for short – to hash messages of arbitrary length. The mode is shown to be (1) *preimage-resistance preserving*, (2) *collision-resistance-preserving* and (3) *indifferentiable* from a random oracle up to $2^{n/2}$ compression function invocations. The fact that many known attacks such as Joux's multicollision, Kelsey-Schneier 2nd preimage and Herding attack do not work on this mode indicates that it may be possible to improve the security bound beyond $2^{n/2}$ compression function invocations. The mode is nearly twice as fast as the Wide pipe mode.

1 Introduction

A hash function $H : \{0,1\}^* \longrightarrow \{0,1\}^n$ is a mathematical function which takes as input a binary string of arbitrary length and outputs a binary string of finite length. A secure hash function can be applied in many applications such as data authentication, digital signature, commitment protocols and password protection. A very popular trend of designing a hash function is executing a *fixed-input-length* (FIL) compression function in a sequential mode as many times as to take the whole message as input. Many practical hash functions, such as MD4 [18], MD5 [19], SHA-0 [16], SHA-1 [17] follow the aforementioned design paradigm. These hash functions precisely have two components: (1) a compression function and (2) a mode of operation. This paper is all about design and analysis of a new hash mode of operation, which we call the *Fast wide pipe* or FWP for short.

Related work. The classical Merkle-Damgärd mode is the most widely used and most studied hash mode of operation. [15, 7]. The mode is simple and collision-resistance-preserving.[1] All the practical hash functions mentioned before are based on the Merkle-Damgärd mode. The landscape is no longer the same. A telltale proof of declining interest of the designers in this mode is that none of the 51 hash functions competing in the ongoing NIST hash function competition uses the classical Merkle-Damgärd mode. The main reasons for discarding this mode by the designers are a few influential attacks: Length extension attack, multi-collision attack [9], Kelsey-Schneier

[1]In a *collision-resistance-preserving* hash function collision resistance of a compression function implies collision resistance of the entire hash function.

2nd preimage attack [12] and Herding attack [10]. On the positive side, the slow and gradual departure of the classical Merkle-Damgärd hash mode has motivated two new lines of research which go nearly hand in hand: (1) design of new modes of operation and (2) development of new security frameworks to analyze hash functions. The first line of research has indeed resulted in a number of new modes of operation – Wide pipe [13], HAIFA [4], Sponge [2], EMD [1] are some of them. One of the major results of the second line of research is the *indifferentiability framework* developed by Maurer *et al.* [14]. Against this framework, we measure the extent to which a hash function is behaving as a random oracle under a suitable assumption on the underlying compression function. Informally speaking if a hash function is *indifferentiable* from a random oracle then, for example, it does not come under length extension attack (assuming the underlying compression function is a FIL random oracle). It is, therefore, important that a new mode of operation is both *collision-resistance-preserving* and *indifferentiable* from a random oracle. Another crucial issue is to recognize that a hash function *indifferentiable* from a random oracle does not guarantee that it is *collision-resistance-preserving* (e.g. modes of operation designed in [6]). These two properties should be analyzed separately [1].

Our contribution. To make a hash function resistant against Joux's multi-collision-type attacks, Lucks has proposed to make the intermediate chaining values of the Merkle-Damgärd mode twice as long as the final hash value; this mode is known as the Wide pipe mode [13]. Suppose the compression function in a Merkle-Damgärd based hash function is defined as $C : \{0,1\}^{m+n} \rightarrow \{0,1\}^n$. Lucks has, very rightly, advocated to use a compression function $C : \{0,1\}^{m+2n} \rightarrow \{0,1\}^{2n}$ to avoid Joux's multi-collision-type attacks [9, 11]. We call this compression function Lucks' compression function. The message and chaining input to the Lucks' compression function are m and $2n$ bits. Using any Lucks' compression function $C : \{0,1\}^{m+2n} \rightarrow \{0,1\}^{2n}$ we design a hash function FWP, where the message and the chaining input to the compression function are $m + n$ and n bits; we, thus, speed up the hashing operation by allowing $m + n$ bits of message instead of just m bits per compression function invocation . At the same time we prove that the FWP mode is *collision-resistance preserving* and *indifferentiable* from a random oracle up to $2^{n/2}$ compression function invocations. The fact that the FWP does not come under Joux's multi-collision-type attacks, such as Kelsey-Schneier 2nd preimage attack, leaves open the possibility to extend the *indifferentiability* bound beyond $2^{n/2}$ compression function invocations.

2 Notation and Convention

Table 1: Notation

$\{0,1\}^{\leq l}$	$\{\varepsilon\} \cup \{0,1\} \cup \{0,1\}^2 \cup \{0,1\}^3 \cup \ldots \cup \{0,1\}^l$
$[x, y]$	The set of integers $x, x+1, \ldots, y$
$a\|\|b$	concatenation of a and b
$\|X\|$	Size of set X; Bit-length if X is a string
$\mathsf{pad}(M)$	The sequence of bits after padding M
fixed-input-length	Fixed input length
variable-input-length	Variable input length
FWP	Fast wide pipe

In addition to the above notation, we shall use another set of notation in the context of indifferentiability results of the hash function FWP. They are described in Sect. 5.1.

3 The New Mode *Fast Wide Pipe* or FWP

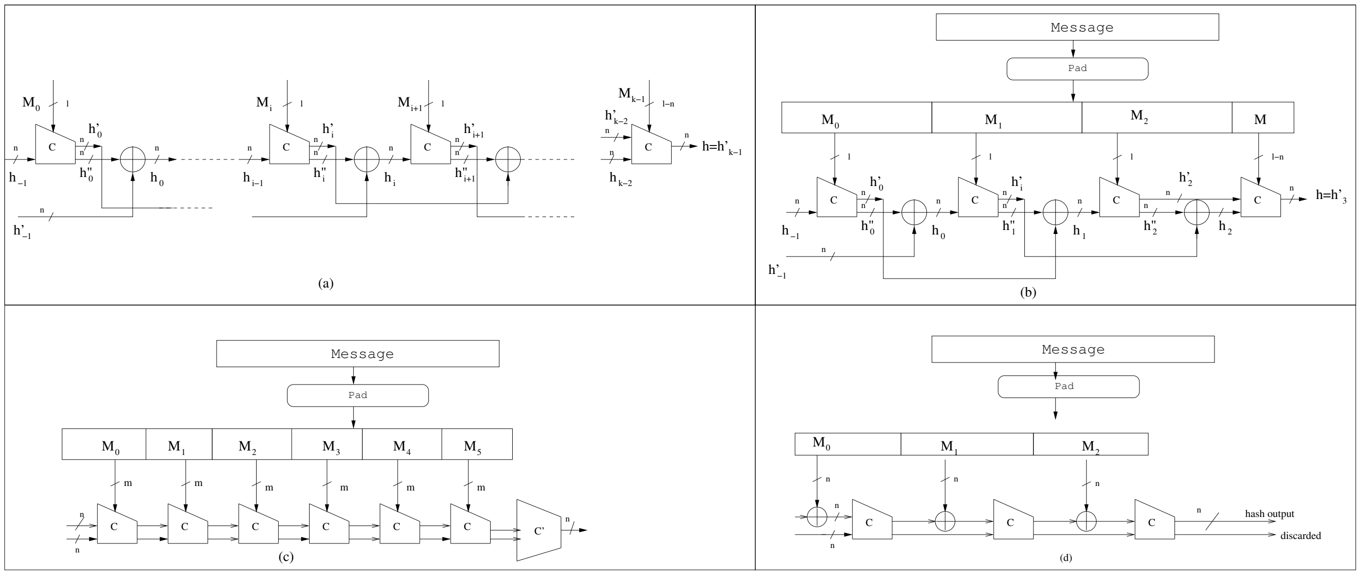


Figure 1: (a) The FWP mode. (b) A four-block example of the FWP mode. (c) The Widepipe mode. (d) The Sponge mode.

In this section we define a new sequential mode of operation *Fast Wide Pipe* (or FWP for short) for hashing messages of length up to 2^{64} bits.

Algorithm 3.1 The FWP mode of operation with the compression function C (*i.e.,* FWP^C)

```
Input: Message $M$
Output: Hash output $h$ of size $n$ bits
Initialize: $h_{-1} = h'_{-1} = 0^n$
1: $M_0||M_1||\dots||M_{k-1} = \mathsf{pad}(M)$ where $|M_i| = l$ for all $i < k-1$ and $|M_{k-1}| = l - n$;
2: $(h_{k-2}, h'_{k-2}) = \text{FWP}_t^C(h_{-1}, h'_{-1}, M_0, M_1, \dots, M_{k-2})$; /* See subfunction below */
3: $C(h_{k-2}||h'_{k-2}||M_{k-1}) = h''_{k-1}||h'_{k-1}$;
4: return hash output $h = h'_{k-1}$;
Subfunction $\text{FWP}_t^C(h_{-1}, h'_{-1}, M_0, M_1, \dots, M_{k-2})$
5: for $i = 0$ to $k-2$ do
6:     $C(h_{i-1}||M_i) = h''_i||h'_i$;
7:     $h_i = h''_i \oplus h'_{i-1}$;
8: end for
9: return $(h_{k-2}, h'_{k-2})$;
```

Diagrammatic representations of the mode are given in Fig. 1(a) and (b). An algorithmic description is in Algorithm 3.1. The padding rule $\mathsf{pad}(M)$ is the execution of the following operation: append t zero bits and a 64-bit encoding of $|M|$ to the message M. Select the least integer $t \geq 0$ such that $|M| + t + n + 64 = 0 \bmod l$. We now make attempts to analyze the security of the

FWP. For the sake of simplicity, we assume $l - n \geq 64$ which ensures that the length-encoding is completely included in the last block. The entire analysis can be modified easily to include the case $l - n < 64$.

4 Security of the FWP: Resistance Against Collision and Preimage Attacks

The main results of this section are two theorems which prove that the collision and the preimage attacks on the FWP mode can be reduced to similar attacks on the underlying compression function (see Algorithm 3.1 for the definition of the FWP mode). In other words, the theorems show that finding collision and preimage on the FWP are at least as hard as finding collision and preimage on the compression function.

Before establishing the security results, we first define the following functions. The functions $C_T, C_B : \{0,1\}^{l+n} \rightarrow \{0,1\}^n$ are defined as $C_T(x) = h'$ and $C_B(x) = h''$ where $C(x) = h''||h'$ (the C is the compression function used by the FWP mode).

Theorem 4.1 *If the compression function C_T is preimage resistant so is the FWP^C.*

PROOF. The theorem can be verified easily by observing the last block of FWP^C. □

Glancing at the XOR operations, one may be tempted to conclude that the FWP may be vulnerable against the generalized birthday attack [20]. The following theorem drives away such fears.

Theorem 4.2 *If the compression function C_T is collision resistant so is the FWP^C.*

PROOF. To prove the theorem we need to prove that, if there exists an adversary who finds a pair of messages (M, M') such that FWP$^C(M)$=FWP$^C(M')$ and $M \neq M'$ then there exists an adversary who can find $X \neq X'$ such that $C_T(X) = C_T(X')$.

Suppose an adversary finds a pair (M, M') such that FWP$^C(M)$=FWP$^C(M')$ and $M \neq M'$. Now there are two possible cases.

CASE 1: $|M| \neq |M'|$. Suppose that the number of message-blocks in $\mathsf{pad}(M)$ and $\mathsf{pad}(M')$ are a and b where $a \neq b$. Note, as per our definition of C and FWP^C, $M_{a-1} \neq M'_{b-1}$ due to the length padding. Now, FWP$^C(M)$=FWP$^C(M')$ implies $C_T(h_{a-2}||h'_{a-2}||M_{a-1}) = C_T(g_{b-2}||g'_{b-2}||M'_{b-1})$. Therefore, we get a collision on C_T.

CASE 2: $|M| = |M'|$. Suppose that the number of message-blocks in $\mathsf{pad}(M)$ is a. Now there are two cases.

CASE 2(A): $C_T(h_{a-2}||h'_{a-2}||M_{a-1}) = C_T(g_{a-2}||g'_{a-2}||M'_{a-1})$
where $h_{a-2}||h'_{a-2}||M_{a-1} \neq g_{a-2}||g'_{a-2}||M'_{a-1}$. Therefore, we obtain a collision on C_T.

CASE 2(B): $C_T(h_{a-2}||h'_{a-2}||M_{a-1}) = C_T(g_{a-2}||g'_{a-2}||M'_{a-1})$
where $h_{a-2}||h'_{a-2}||M_{a-1} = g_{a-2}||g'_{a-2}||M'_{a-1}$.

The above equation implies that FWP$^C_t(0^n||0^n||M_0||\ldots||M_{a-2})$=FWP$^C_t(0^n||0^n||M'_0||\ldots||M'_{a-2})$ which in turn implies collision on C_T by Lemma 4.3 (the definition of FWP^C_t is provided in Algorithm 3.1). Now the only remaining part needed to complete the proof is the proof of Lemma 4.3 which is provided below. □

The following lemma has been used in Theorem 4.2. It will be further used to obtain some indifferentiability results of the FWP^C in Sect. 5.

Lemma 4.3 *If the compression function C_T is collision resistant then the FWP_t^C is free-start collision resistant for fixed length messages. In other words, if there exists an adversary who finds two triples $(h_{-1}, h'_{-1}, M) \neq (g_{-1}, g'_{-1}, M')$ such that $|M| = |M'|$ ($|M|$ is a multiple of l) and $FWP_t^C(h_{-1}, h'_{-1}, M) = FWP_t^C(g_{-1}, g'_{-1}, M')$, then there exists an adversary who finds $X \neq X'$ such that $C_T(X) = C_T(X')$.*

Proof. Suppose there exists an adversary who finds two triples $(h_1, h'_1, M) \neq (g_1, g'_1, M')$ such that $|M| = |M'|$ (the number of message-blocks in M is a) and $\mathrm{FWP}_t^C(h_{-1}, h'_{-1}, M) = \mathrm{FWP}_t^C(g_{-1}, g'_{-1}, M')$. In order to obtain a pair $X \neq X'$ such that $C_T(X) = C_T(X')$ we need to check at most a equations whether they are satisfied:

$$C(h_{i-1}, M_i) \stackrel{?}{=} C(g_{i-1}, M'_i) \text{ where } i = a-1, \ldots, 0.$$

We claim that the above verification will produce some m with $0 \leq m \leq a-1$ such that $C_T(h_{m-1}, M_m) = C_T(g_{m-1}, M'_m)$ and $(h_{m-1}, M_m) \neq (g_{m-1}, M'_m)$ and thus, the lemma is proved. This claim can be proved by the following crucial observation on FWP_t^C.
Observation: For all $i \in [0, a-1]$, $(h_i, h'_i) = (g_i, g'_i)$ implies one of the following two statements:
(1) $(h_{i-1}, M_{i-1}) \neq (g_{i-1}, M'_{i-1})$ which implies collision on C_T,
(2) $(h_{i-1}, M_{i-1}) = (g_{i-1}, M'_{i-1})$ which implies $(h_{i-1}, h'_{i-1}) = (g_{i-1}, g'_{i-1})$. □

Next, we move on to analyze the FWP in a different security framework known as the indifferentiability framework.

5 Security of the FWP Mode: Indifferentiable From a Random Oracle

In this section we discuss the indifferentiability property of the FWP mode. In the context of hash function, an important use of the indifferentiability framework developed by Maurer *et al.* [14] is the determination of whether a *variable-input-length* hash function behaves reasonably randomly when the underlying compression function is a *fixed-input-length* random oracle. There is a considerable chance for the reader to be lured into believing that the collision resistance preservation (described in Sect. 4) and the indifferentiability property of a hash function may be related. In particular, one may be inclined to intuiting that one property implies the other. Such intuition is not true [1]. These two properties are orthogonal and need to be analyzed separately.

5.1 Preliminaries: Introduction to Indifferentiability Framework

We begin with the definition of a random oracle; this useful object will be used frequently in the subsequent discussion.

Definition 5.1 (Random oracle) *A random oracle is a function $\mathsf{RO} : X \to Y$ chosen uniformly at random from the set of all $|Y|^{|X|}$ functions that map $X \to Y$. In other words, a function $\mathsf{RO} : X \to Y$ is a random oracle if and only if, for each $x \in X$, the $\mathsf{RO}(x)$ is chosen uniformly at random from Y.*

Corollary 5.2 *If a function $\mathsf{RO} : X \to Y$ is a random oracle then*

$$\Pr[\mathsf{RO}(x) = y | \mathsf{RO}(x_1) = y_1,\ \mathsf{RO}(x_2) = y_2,\ \ldots,\ \mathsf{RO}(x_q) = y_q] = \frac{1}{|Y|}$$

where $x \notin \{x_1, x_2, \ldots, x_q\}$, $y \in Y$ and $q \in \mathbb{Z}$.

Now we introduce the indifferentiability framework and briefly discuss its significance. The following definition is a slightly modified version of the original definition provided in [14, 6].

Definition 5.3 (Indifferentiability framework) *[14] A Turing machine T with oracle access to an ideal primitive $\mathcal{F}$ is said to be $(t_\mathcal{A}, t_S, q, \sigma, \varepsilon)$-indifferentiable from an ideal primitive $\mathcal{G}$ if there exists a simulator S such that for any distinguisher $\mathcal{A}$ the following equation is satisfied:*

$$\mathbf{Adv}_\mathcal{A}((T,\mathcal{F}),(\mathcal{G},S)) = |\Pr[\mathcal{A}^{T,\mathcal{F}} = 1] - \Pr[\mathcal{A}^{\mathcal{G},S} = 1]| < \varepsilon$$

The simulator S is an interactive algorithm which has oracle access to $\mathcal{G}$ and runs in time at most t_S. The distinguisher $\mathcal{A}$ runs in time at most $t_\mathcal{A}$ and makes at most q queries. The total message blocks queried by $\mathcal{A}$ is at most σ.

Question 1 *What is the significance of indifferentiability property?*

Answer. Suppose, an ideal primitive $\mathcal{G}$ (e.g. a *variable-input-length* random oracle) is indifferentiable from an algorithm T based on another ideal primitive $\mathcal{F}$ (e.g. a *fixed-input-length* random oracle). In such case, any cryptographic system $\mathcal{P}$ based on $\mathcal{G}$ is as secure as the $\mathcal{P}$ based on $T^\mathcal{F}$ (i.e., $T^\mathcal{F}$ replaces $\mathcal{G}$ in $\mathcal{P}$). See [14] for more on that. □

Pictorial Description of Def. 5.3(Fig. 2). In the figure, five entities involved in Def. 5.3 are shown with an example. Suppose, the oracle Turing machine T, the ideal primitives $\mathcal{F}$, $\mathcal{G}$ are, respectively, a hash function H, random oracles ro and RO. The exchange of queries and responses is also shown in the figure. Note that it is forbidden to issue queries in the opposite directions. For example, the hash function H can send a query to ro and receive response, but never the other way round. In this setting, Def. 5.3 addresses the degree to which any computationally bounded adversary is unable to distinguish between Option 1 and Option 2. □

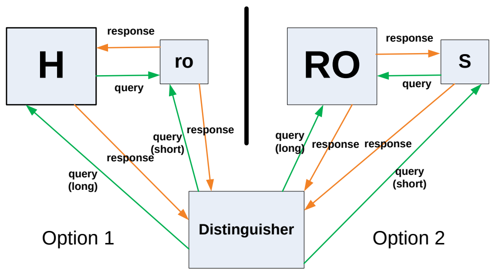
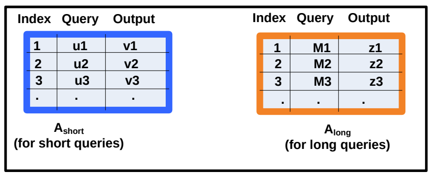


Figure 2: The entities and their behavior involved in the *indifferentiability* framework of Def. 5.3; $T \equiv H$, $\mathcal{F} \equiv \mathsf{ro}$, $\mathcal{G} \equiv \mathsf{RO}$, $S \equiv$ simulator (see description above). In Sect. 5.2, H is the FWP hash function.



Index	Query	Output
1	u1	v1
2	u2	v2
3	u3	v3
.	.	.

A_{short} (for short queries)

Index	Query	Output
1	M1	z1
2	M2	z2
3	M3	z3
.	.	.

A_{long} (for long queries)

Figure 3: Several databases maintained by the distinguisher

5.2 Indifferentiability Framework for FWP: Designing a Simulator S

In this section we describe the entities of Fig. 2 with respect to the hash function FWP: $\{0,1\}^{\le 2^{64}} \to \{0,1\}^n$. The mode FWP is defined in Fig. 4. In the rest of the paper the H is understood to be the FWP hash function. The *fixed-input-length* random oracle $\mathsf{ro} : \{0,1\}^{r+n} \to \{0,1\}^{2n}$ is the compression function invoked by the FWP mode. The *variable-input-length* random oracle RO is

defined as $\mathsf{RO} : \{0,1\}^{\leq 2^{64}} \rightarrow \{0,1\}^n$. The only remaining part to complete the indifferentiability framework is designing a simulator S. This section is devoted to that. The fifth entity of Fig. 2, which is an arbitrary distinguisher $\mathcal{A}$, is discussed in Sect. 5.3. We kick off with the notation.
Notation. Table 2 provides the notation useful to follow our *indifferentiability* results on the new hash function FWP. Note that the notation can be very easily adapted to any hash function based on a sequential mode of operation. □

Table 2: The notation used in the *indifferentiability* framework for FWP (see Fig. 4)

Symbol	bit-length	Description
$\mathsf{A}_{\text{short}}$, A_{long}	-	Current query-response arrays
$\mathsf{A}_{\text{inter}}$	-	Array for intermediate query-responses
$A[i, i-1, \ldots j]$	-	Array (or bit-string) A truncated between indices i and j
$\mathcal{A}$	-	A distinguisher
$\mathcal{A}'$	-	Modification of the distinguisher $\mathcal{A}$
$\ell(M)$	-	Number of compression function calls to hash M
λ	0	Empty String
M	$\leq 2^{64}$	Message $M = m^1 m^2 \ldots m^{\ell(M)}$
m^k, $m^{\ell(M)}$	$r, r-n$	Messages of kth and $\ell(M)$th compression functions ($k < m^{\ell(M)}$)
$\mathsf{MesgVer}$	-	Message verification algorithm
$\mathsf{MesgRecon}$	-	Message reconstruction algorithm
q, σ	-	Maximum number of queries and blocks used by distinguisher
ro, RO	-	Random oracles
$\mathcal{S}$	-	Set of reconstructed messages given a *short query*
S	-	The simulator
$t_{\mathcal{A}}$, t_S	-	Time of $\mathcal{A}$ and S
$u^{k\prime}$	n	Chaining input to kth compression function ($k < m^{\ell(M)}$)
$u^{\ell(M)\prime\prime} \| u^{\ell(M)\prime}$	$2n$	Chaining input to $\ell(M)$th compression function
u^k, $u^{\ell(M)}$	$r+n, r+n$	Total input to kth and $\ell(M)$th compression functions
$v^{k\prime}$, $v^{k\prime\prime}$	n, n	Two halves of output from kth compression function
v^k	$2n$	Total output from kth compression function
z	n	Final hash value

Now we define a few terms – in relation to Fig. 4 and 2 – which will be used to arrive at our main indifferentiability results of Sect. 5.3. **Queries and lists.** We now define various types of queries and lists (or arrays) that can potentially be used by a distinguisher to separate a hash function from a random oracle. The first assumption is that a distinguisher does not resubmit to an oracle a query whose response is already known. This is a valid assumption because, in our case, an identical oracle – any of FWP hash function, ro, RO and S of Fig. 2 – gives identical response to an identical query (it would be further clear when we shall concretely define the simulator S). Our next assumption is that, unless otherwise specified, a query is known to be submitted by the distinguisher. In the present case, we are not interested in queries submitted by the simulator S or by the hash function FWP. Now we define two special types of queries.

Definition 5.4 (*Short* and *long* query) *A query submitted to S or ro is defined as a* short *query. Similarly, a query submitted to FWP or RO is defined as the* long *query (see Fig. 2).*

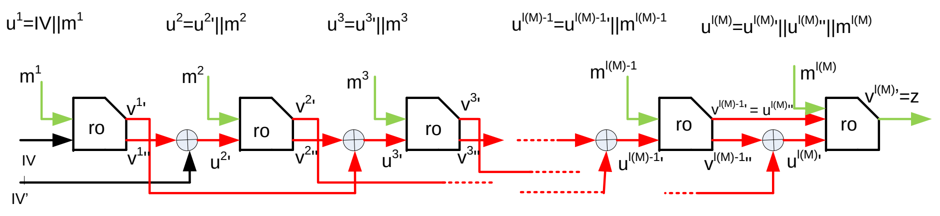


Figure 4: Message $M = m^1m^2 \ldots m^{\ell(M)}$ is hashed by FWP^{ro}. The symbols are described in Table 2.

At this time it is important to discuss a subclass of *short* and *long* queries known as *trivial* queries. For easy understanding, we try to introduce the notion without the rigors of mathematical notation as much as possible; however, our treatment is logically sound and foolproof. The motivation behind the determination of *trivial* queries is that their outputs are implied by the previous queries and their responses, no matter whether the distinguisher is interacting with Option 1 or Option 2 of Fig. 2. Therefore, *trivial* queries cannot be used to distinguish between two systems, even if they satisfy specific 'bad' conditions. Before we formally define *trivial* queries, some discussion on the databases maintained by the distinguisher and two special functions MesgVer and MesgRecon are necessary. We first discuss them briefly.

Databases of the distinguisher. Let us assume that a distinguisher uses two arrays: (1) $\mathsf{A}_{\text{short}}$ for storing *short* queries and the responses, and (2) A_{long} for *long* queries and the responses (see Fig. 3). Queries and their responses are indexed by the time they are submitted. Note that the simulator S can access $\mathsf{A}_{\text{short}}$ but not A_{long}.

Algorithm 5.1 Message verification algorithm $\mathsf{MesgVer}(\cdot,\cdot)$

```
Input:  Array $\mathsf{A}_{\text{short}}$, bit-string $M$ ($|M| \leq 2^{64}$)
Output:  A bit $b$
(See Table 2 and Fig. 4 for the notation.)
 1: Set $b = 1$;
 2: for $i = 1$ to $\ell(M)$ do
 3:    Compute $u^i$ from $m^i$, $v^{i-1}$, $v^{i-2}$;
 4:    if $\nexists v$ such that $(u^i, v) \in \mathsf{A}_{\text{short}}$ then
 5:       return $b = 0$;
 6:    else
 7:       Compute $v^i$ using $u^i$ and $\mathsf{A}_{\text{short}}$;
 8:    end if
 9: end for
10: return $b$;
```

Discussion on algorithms MesgVer and MesgRecon. Informally speaking, MesgVer is a function which takes two inputs – the current list $\mathsf{A}_{\text{short}}$, a *long query* M – to verify whether the *long query* M is a valid message for the hash mode FWP. What it essentially does is compute all compression function inputs – u^1, u^2, ..., $u^{\ell(M)}$ – sequentially and checks whether they exist in $\mathsf{A}_{\text{short}}$. The MesgVer algorithm has been described in Algorithm 5.1.

The MesgRecon algorithm, in some sense, works in the opposite direction. It takes the current list $\mathsf{A}_{\text{short}}$ and a *short query* x as inputs and reconstructs a set of messages $\mathcal{S}$ such that each message $M \in \mathcal{S}$ is a valid message for FWP mode and, moreover, the input to the last compression function

Algorithm 5.2 Message reconstruction algorithm $\mathsf{MesgRecon}(\cdot,\cdot)$

Input: Array $\mathsf{A}_{\text{short}}$, bit-string x ($|x| = r + n$)

Output: A set of reconstructed messages $\mathcal{S}$

Assumption: For simplicity we assume $r - n \geq 64$. This makes 64-bit length-encoding in the last message block. If $r - n < 64$ then we need more than one block to determine the length.

(See Table 2 and Fig. 4 for the notation.)

```
1: Compute $\ell(M)$ from $x[64, \ldots, 2, 1]$;
2: Break $x = u^{\ell(M)'} || v^{\ell(M)-1'} || m^{\ell(M)}$ such that $v^{\ell(M)-1'} = x[r, r-1, \ldots, r-n+1]$;
3: Construct $G = \{(u, v) \in \mathsf{A}_{\text{short}} \mid v[n, \ldots, 2, 1] = v^{\ell(M)-1'}\}$;
4: if $|G| \neq 1$ then
5:   return $\mathcal{S} = \emptyset$;
6: end if
7: for $i = \ell(M) - 1$ to 1 do
8:   $m^i = u[r, \ldots, 2, 1]$;
9:   Compute $v^{i-1'} = u^{i+1'} \bigoplus v^{i''}$;
10:   if $i \neq 1$ then
11:     Construct $G = \{(u, v) \in \mathsf{A}_{\text{short}} \mid v[n, \ldots, 2, 1] = v^{i-1'}\}$;
12:     if $|G| \neq 1$ then
13:       return $\mathcal{S} = \emptyset$;
14:     end if
15:   else
16:     if $u[r+n, \ldots, r+1] = IV$ and $v^{i-1'} = IV'$ then
17:       Compute $M = m^1 m^2 \ldots m^{\ell(M)}$;
18:       return $\mathcal{S} = \{M\}$;
19:     else
20:       return $\mathcal{S} = \emptyset$;
21:     end if
22:   end if
23: end for
```

is x. The algorithm is described in Algorithm 5.2. □

Now we are ready to define the *trivial queries.*

Definition 5.5 (Trivial short query) *A* short query x *is a* trivial short query *if the following conditions hold:*

- $\mathsf{MesgRecon}(\mathsf{A}_{short},\, x) = \{M\}$.
- *The M has been queried previously as a* long *query (i.e. $\exists v$ such that $(M, v) \in \mathsf{A}_{long}$).*

Definition 5.6 (Trivial long query) *A* long query M *is a* trivial long query *if the following conditions hold:*

- *$\mathsf{MesgVer}(\mathsf{A}_{short},\, M)$=1. Suppose the final input $u^{\ell(M)}$ computed in $\mathsf{MesgVer}(\mathsf{A}_{short},\, M)$ is the ith query in A_{short}.*
- $\mathsf{MesgRecon}(\mathsf{A}_{short}[i-1, \ldots, 2, 1], u^{\ell(M)}) = \{M\}$.

The *nontrivial short* and *long* queries are obvious from the above definitions.

Definition 5.7 (Nontrivial queries) *A* short query x *is a* nontrivial short query *if it is not a* trivial short *query. Similarly, a* long query M *is a* nontrivial long query *if it is not a* trivial long *query.*

At this point it is useful to, once more, remember the motivation behind separating the trivial queries from all queries. The distinguisher may communicate with $(\mathsf{FWP}, \mathsf{ro})$ or (RO, S). Irrespective of whether it is communicating with $(\mathsf{FWP}, \mathsf{ro})$ or (RO, S), the responses of the trivial queries should be implied by the previous query-responses. Therefore, the trivial queries do not help a distinguisher to differentiate between $(\mathsf{FWP}, \mathsf{ro})$ and (RO, S) (see Fig. 2). We have just concretely defined the trivial queries in Def. 5.5and 5.6. However, we still cannot say whether the *trivial* queries indeed fulfil the motivation until we prove the existence of a compatible simulator. Such a simulator S is described below.

Algorithm 5.3 The simulator $S(\cdot)$

```
Input: short query x
Output: 2n-bit string v
1: $\mathcal{S}$=MesgRecon($\mathsf{A}_{\text{short}}$, $x$);
2: if $|S| = 1$ then
3:     return $v = \mathsf{RO}(M)$; /* $\mathcal{S} = \{M\}$ */
4: end if
5: return $v = \mathsf{ro}(x)$;
```

Our design of indifferentiability framework is now complete, except establishing a property that shows, under trivial queries, both $(\mathsf{FWP},\ \mathsf{ro})$ and $(\mathsf{RO},\ S)$ behave identically, if they are supplied with identical $\mathsf{A}_{\text{short}}$ and A_{long}. We capture this property in the following lemma.

Lemma 5.8 *Suppose, for a distinguisher $\mathcal{A}$, the lists A_{short} and A_{long} are identical for both (FWP, ro) and (RO, S) after the ith query. Then the following statements are true.*

1. *If M is the $(i+1)$th* trivial long query *then $FWP^{\mathsf{ro}}(M) = \mathsf{RO}(M)$.*
2. *If x is the $(i+1)$the* trivial short query *then $S(x) = \mathsf{ro}(x)$.*

Proof. The proof is immediate from the construction of the simulator S which is described in Algorithm 5.3. □

5.3 Bounding the Advantage of an Arbitrary Distinguisher

After designing the simulator S in the previous section, now we are left with the most important part of the paper: to compute an ε as a function of $(t_{\mathcal{A}}, t_S, q, \sigma)$ (see Def. 5.3). To that end, we first design an arbitrary oracle algorithm $\mathcal{A}$ (see Algorithm 5.4) that separates $(\text{FWP}, \mathsf{ro})$ from (RO, S). Algorithm 5.4 is characterized by two functions: (1) the $f_{query}(\cdot,\cdot)$ which computes the

Algorithm 5.4 An arbitrary distinguisher $\mathcal{A}(\cdot,\cdot)$ telling apart $(\text{FWP}, \mathsf{ro})$ and (RO, S)

```
Input:  An oracle $\mathcal{O}_{small} : \{0,1\}^{r+n} \rightarrow \{0,1\}^{2n}$ /* $\mathcal{O}_{small}$ is either ro or $S$ */
        An oracle $\mathcal{O}_{big} : \{0,1\}^{\leq 2^{64}} \rightarrow \{0,1\}^{n}$ /* $\mathcal{O}_{big}$ is either FWP or RO */
Output: A bit $b$
1:  Initialize: $\mathsf{A}_{\text{short}}$, $\mathsf{A}_{\text{long}} = \emptyset$;
2:  for $i = 1$ to $q$ do
3:    $(X_i, tag) = f_{query}(\mathsf{A}_{\text{short}}, \mathsf{A}_{\text{long}})$; /* $tag = 0, 1$ implies long, short queries */
4:    if $tag = 0$ then
5:      $M_i = X_i$, $z_i \longleftarrow \mathcal{O}_{big}(M_i)$;
6:      $\mathsf{A}_{\text{long}} = \mathsf{A}_{\text{long}} \cup \{(M_i, z_i)\}$; /* Updating $\mathsf{A}_{\text{long}}$ */
7:      $b = f_{cond}(\mathsf{A}_{\text{short}}, \mathsf{A}_{\text{long}})$;
8:      if $b = 1$ then
9:        return $b$; /* The system is (FWP, ro) */
10:     end if
11:   end if
12:   if $tag = 1$ then
13:     $x_i = X_i$, $y_i \longleftarrow \mathcal{O}_{small}(x_i)$;
14:     $\mathsf{A}_{\text{short}} = \mathsf{A}_{\text{short}} \cup \{(x_i, y_i)\}$; /* Updating $\mathsf{A}_{\text{short}}$ */
15:     $b = f_{cond}(\mathsf{A}_{\text{short}}, \mathsf{A}_{\text{long}})$;
16:     if $b = 1$ then
17:       return $b$; /* The system is (FWP, ro) */
18:     end if
19:   end if
20: end for
21: return $b = 0$; /* The system is $(\mathsf{RO}, S)$ */
```

next query, and (2) the $f_{cond}(\cdot,\cdot)$ which decides whether the system is (FWP, ro) or (RO, S). Both the functions take the arrays $\mathsf{A}_{\text{short}}$, A_{long} as inputs. To bound the advantage of $\mathcal{A}$, we slightly modify $\mathcal{A}$ to design $\mathcal{A}'$ which is described in Algorithm 5.5. We now discuss the algorithms briefly.

Discussion on Algorithm 5.4 and 5.5. Both $\mathcal{A}$ and $\mathcal{A}'$ have identical query function f_{query}. We only modify f_{cond} of $\mathcal{A}$ to design f'_{cond} of $\mathcal{A}'$. The additional parts of $\mathcal{A}'$ are placed within boxes in Algorithm 5.5. The algorithm $\mathcal{A}'$, in addition to $\mathsf{A}_{\text{short}}$ and A_{long}, uses an extra array $\mathsf{A}_{\text{inter}}$ which, using a function $\mathsf{MesgDecom}(M_i)$, stores all intermediate inputs and outputs for any *long query* M_i applied to FWP. Our main task is to define a suitable f'_{cond} such that the following inequality holds:

$$\max_{\mathcal{A}} |\Pr[\mathcal{A}(\text{FWP}, \mathsf{ro}) = 1] - \Pr[\mathcal{A}(\mathsf{RO}, S) = 1]| \leq \max_{\mathcal{A}'} \Pr[\mathcal{A}'(\text{FWP}, \mathsf{ro}) = 1] \quad (1)$$

where the maximum values of the right hand side and the left hand side are based on the suitable choices of (1) f_{query} and f_{cond}, and (2) f'_{cond} respectively. It is easy to show that he above inequality

Algorithm 5.5 Algorithm $\mathcal{A}'(\cdot,\cdot)$ computing Bad events

```
Input:  An oracle $\mathcal{O}_{small} : \{0,1\}^{r+n} \rightarrow \{0,1\}^{2n}$, /* $\mathcal{O}_{small}$ is $\mathsf{ro}$ */
        An oracle $\mathcal{O}_{big} : \{0,1\}^{\leq 2^{64}} \rightarrow \{0,1\}^{n}$ /* $\mathcal{O}_{big}$ is FWP */
Output: A bit $b$
1:  Initialize: $\mathsf{A}_{\text{short}}$, $\mathsf{A}_{\text{long}}$= $\emptyset$, $\boxed{\mathsf{A}_{\text{inter}} = \emptyset,}$ $\mathsf{Bad}$=0;
2:  for $i = 1$ to $q$ do
3:    $(X_i, tag) = f_{query}(\mathsf{A}_{\text{short}}, \mathsf{A}_{\text{long}})$; /* $tag = 0, 1$ implies long, short queries */
4:    if $tag = 0$ then
5:      $M_i = X_i$, $z_i \longleftarrow \mathcal{O}_{big}(M_i)$;
6:      $\mathsf{A}_{\text{long}}$=$\mathsf{A}_{\text{long}} \cup \{(M_i, z_i)\}$; /* Updating $\mathsf{A}_{\text{long}}$ */
7:      $\boxed{\mathsf{A}_{\text{inter}} = \mathsf{A}_{\text{inter}} \cup \mathsf{MesgDecom}(M_i);}$ /* Updating $A_{inter}$ */
8:      $\boxed{b = f'_{cond}(\mathsf{A}_{\text{short}}, \mathsf{A}_{\text{long}}, A_{inter});}$ /* Checking condition for $\mathsf{Bad}$ event */
9:      if $b = 1$ then
10:       return $b$; /* $\mathsf{Bad}$ event */
11:     end if
12:   end if
13:   if $tag = 1$ then
14:     $x_i = X_i$, $y_i \longleftarrow \mathcal{O}_{small}(x_i)$;
15:     $\mathsf{A}_{\text{short}}$=$\mathsf{A}_{\text{short}} \cup \{(x_i, y_i)\}$; /* Updating $\mathsf{A}_{\text{short}}$ */
16:     $\boxed{b = f'_{cond}(\mathsf{A}_{\text{short}}, \mathsf{A}_{\text{long}}, A_{inter});}$ /* Checking condition for $\mathsf{Bad}$ event */
17:     if $b = 1$ then
18:       return $b$; /* $\mathsf{Bad}$ event */
19:     end if
20:   end if
21: end for
22: return $b = 0$; /* $\mathsf{Good}$ event */
```

implies $\mathbf{Adv}_{\mathcal{A}}((\mathrm{FWP},\mathsf{ro}),(\mathsf{RO},S)) \leq \max_{\mathcal{A}} \Pr[\mathcal{A}'(\mathrm{FWP},\mathsf{ro}) = 1]$. We now define a suitable f'_{cond} recursively.

Definition 5.9 (f'_{cond} of Algorithm 5.5) *The definition is divided into two complementary parts. (1) Let the ith query computed by f_{query} of $\mathcal{A}'$ be a* nontrivial long query *denoted by M_i. Then $f'_{cond} = 1$ if one or more following conditions are satisfied.*

- *Collision between the final input for the current* long query *M_i and the final input for some previous* long query *M_j. That is, $u_i^{\ell(M_i)} = u_j^{\ell(M_j)}$ for some $j < i$.*
- *Collision between the final input for the current* long query *M_i and some intermediate input for some previous* long query *M_j. That is, $u_i^{\ell(M_i)} = u_j^k$ for some $j \leq i$ and $k < \ell(M_j)$.*
- *Collision between some intermediate input for the current* long query *M_i and the final input for some previous* long query *M_j. That is, $u_i^k = u_j^{\ell(M_j)}$ for some $j < i$ and $k < \ell(M_i)$.*
- *Collision between the final input for the current* long query *M_i and some previous* short query *x_j. That is, $u_i^{\ell(M_i)} = x_j$ for some $j < i$.*

Otherwise $f'_{cond} = 0$.
(2) Let the ith query computed by f_{query} of $\mathcal{A}'$ be a nontrivial short query *denoted by x_i. Then $f'_{cond} = 1$ if the following condition is satisfied.*

- *Collision between the current* short query *x_i and the final input for some previous* long query *M_j. That is, $x_i = u_j^{\ell(M_j)}$ for some $j < i$.*

Otherwise $f'_{cond} = 0$.

Now we state the following theorem.

Theorem 5.10 *Under Def. 5.9 of f'_{cond} the following inequality holds.*

$$\mathbf{Adv}_{\mathcal{A}}((FWP,\mathit{ro}),(\mathit{RO},S)) \leq \max_{\mathcal{A}'} \Pr[\mathcal{A}'(FWP,\mathit{ro}) = 1].$$

Proof. The theorem has been proved for a general domain extension in [3]. Note that the event $\mathcal{A}'(\mathrm{FWP},\mathsf{ro}) = 1$ is also an event that $\mathcal{A}(\mathrm{FWP},\mathsf{ro})$ invokes as defined in Def. 5.9 which has been termed a Bad event for a GDE. So by using Theorem 1 of [3] we have our result. □

In the remainder of the section we strive to obtain an upper bound ε on $\max_{\mathcal{A}'} \Pr[\mathcal{A}'(\mathrm{FWP},\mathsf{ro}) = 1]$. According to Theorem 5.10, ε is an upper bound on $\mathbf{Adv}_{\mathcal{A}}((\mathrm{FWP},\mathsf{ro}),(\mathsf{RO},S))$ too.

We have two databases $\mathsf{A}_{\text{short}}$ and $\mathsf{A}_{\text{inter}}$ which essentially store all invocations of ro. Each element of $\mathsf{A}_{\text{short}}$ and $\mathsf{A}_{\text{inter}}$ is of the form (u, v) where $u \in \{0,1\}^{r+n}$ and $v \in \{0,1\}^{2n}$. We denote the ith pair by $\mathsf{A}_{\text{short}}(i) = (\mathsf{A}_{\text{short}}(i,1), \mathsf{A}_{\text{short}}(i,2))$ and $\mathsf{A}_{\text{inter}}(i) = (\mathsf{A}_{\text{inter}}(i,1), \mathsf{A}_{\text{inter}}(i,2))$.

Whenever we add a pair (u, v) in $\mathsf{A}_{\text{inter}}$ it corresponds to a pair (M, i) such that when we compute $FWP^{\mathsf{ro}}(M)$, the ith intermediate input, output are u and v respectively. Note that $i = \ell(M)$ and in that case $FWP^{\mathsf{ro}}(M) = v[2n, 2n-1, \ldots n+1]$.

We define the following bad events. It mainly considers either the unexpected collisions in the first or last half of the outputs of ro which are stored in one of the two databases $\mathsf{A}_{\text{short}}$ and $\mathsf{A}_{\text{inter}}$ during query-responses of $\mathcal{A}'$ or collision on least significant n-bit inputs of ro stored in $\mathsf{A}_{\text{inter}}$ with least significant n-bit inputs of ro stored in any one of the two lists.

1. $\mathsf{A}_{\text{short}}$ vs $\mathsf{A}_{\text{short}}$ for output collision: If $\mathsf{A}_{\text{short}}(i, 2)[n, n-1, \ldots 1] = \mathsf{A}_{\text{short}}(i', 2)[n, n-1, \ldots 1]$ or $\mathsf{A}_{\text{short}}(i, 2)[2n, 2n-1, \ldots n+1] = \mathsf{A}_{\text{short}}(i', 2)[2n, 2n-1, \ldots n+1]$ for some $i \neq i'$. We call it type-1 bad.

2. $\mathsf{A}_{\text{short}}$ vs $\mathsf{A}_{\text{inter}}$ for output collision: If $\mathsf{A}_{\text{short}}(i, 2)[n, n-1, \ldots 1] = \mathsf{A}_{\text{inter}}(i', 2)[n, n-1, \ldots 1]$ or $\mathsf{A}_{\text{short}}(i, 2)[2n, 2n-1, \ldots n+1] = \mathsf{A}_{\text{inter}}(i', 2)[2n, 2n-1, \ldots n+1]$ for some i, i' such that the following is not true:

   $\mathsf{A}_{\text{inter}}(i', 2)$ corresponds to the pair (M, j) and the computation of $FWP^{\mathsf{ro}}(M)$ up to $j-1$ intermediate input is already in the list $\{\mathsf{A}_{\text{short}}(r) : r \leq j-1\}$ and the j^{th} intermediate input is $\mathsf{A}_{\text{inter}}(i', 2)$.

   We call it type-2 bad.

3. $\mathsf{A}_{\text{inter}}$ vs $\mathsf{A}_{\text{inter}}$ for output collision: If $\mathsf{A}_{\text{short}}(i, 2)[n, n-1, \ldots 1] = \mathsf{A}_{\text{inter}}(i', 2)[n, n-1, \ldots 1]$ or $\mathsf{A}_{\text{short}}(i, 2)[2n, 2n-1, \ldots n+1] = \mathsf{A}_{\text{inter}}(i', 2)[2n, 2n-1, \ldots n+1]$ for some i, i' such that the pairs corresponding to $\mathsf{A}_{\text{short}}(i, 2)$ and $\mathsf{A}_{\text{inter}}(i', 2)$ are not identical. We call it type-3 bad.

4. $\mathsf{A}_{\text{inter}}$ vs both list for input collision: $\mathsf{A}_{\text{inter}}(i, 1)[n, n-1, \ldots 1] = \mathsf{A}_{\text{inter}}(i', 1)[n, n-1, \ldots 1]$ or $\mathsf{A}_{\text{inter}}(i, 1)[n, n-1, \ldots 1] = \mathsf{A}_{\text{short}}(j, 1)[n, n-1, \ldots 1]$ for some $i \neq i'$. We call it type-4 bad.

Lemma 5.11 *If f'_{cond} (see definition 5.9) returns one then one of the four types of bad events would occur.*

Proof. The proof is immediate. □

Note that for a short query we add one element in $\mathsf{A}_{\text{short}}$ and for a long query we add $\ell = \ell(M)$ elements in A_{int}. In total we update σ elements in two databases after q queries where σ is the total number of blocks in all q queries (both short and long). We define bad^i to be one of the bad event when we add i^{th} element, $1 \leq i \leq \sigma$. The complement of the event is denoted by good^i. We estimate the following probability for different possible cases:

$$\Pr[\mathsf{bad}^i | \wedge_{j=1}^{i-1} \mathsf{good}^j].$$

We divide into two cases based on i^{th} update (u, v) is on $\mathsf{A}_{\text{short}}$ or $\mathsf{A}_{\text{inter}}$.

- Case 1. Bad event on the update of $\mathsf{A}_{\text{short}}$: It can happen in two ways. Either the adversary correctly guesses u which appear before in $\mathsf{A}_{\text{inter}}$ or the outputs collide accidentally with one of the previous outputs stored in $\mathsf{A}_{\text{short}}$ or $\mathsf{A}_{\text{inter}}$ given that the guess is not correct. Note that if the guess is not correct then the input u is fresh and its output is uniformly distributed. The collision occur in one of the n-bits with probability at most $2(i-1)/2^n$. Moreover, if u appears as j^{th} intermediate input of $FWP^{\mathsf{ro}}(M)$ for some M such that (M, j) corresponded by an element of $\mathsf{A}_{\text{inter}}$ then the type-4 bad event occur with probability $(i-1)/2^n$.

  Now given that good event, all information to $\mathcal{A}$ so far, is independent with the internal computation. So the guess is correct with some of internal input has probability bounded by $(i-1)/2^n$. So
  $$\Pr[\mathsf{bad}^i | \wedge_{j=1}^{i-1} \mathsf{good}^j] \leq 4(i-1)/2^n.$$

- Case 2. Bad event on the update of $\mathsf{A}_{\mathrm{inter}}$: This probability can be bounded by random oracle collision probability as the input u freshly appear due to the good event. The following can be shown easily:

  $\Pr[type-4\ \mathsf{bad}^i|\wedge_{j=1}^{i-1}\mathsf{good}^j] \leq (i-1)/2^n, \Pr[[]type-2 \text{ or } 3\ \mathsf{bad}^i|\wedge_{j=1}^{i-1}\mathsf{good}^j] \leq 2(i-1)/2^n$

  and hence $\Pr[\mathsf{bad}^i|\wedge_{j=1}^{i-1}\mathsf{good}^j] \leq 3(i-1)/2^n$.

Combining all these cases we obtain the probability of bad event is at most $\sigma(\sigma-1)/2^{n-1}$. Now we state our indifferentiability results.

Theorem 5.12 *The FWP hash is* $(t_{\mathcal{A}}, t_S, q, \sigma, \varepsilon^*)$*-indifferentiable in the random oracle model for the compression function, for any* $t_{\mathcal{A}}$*, with* $t_S = \ell \cdot \mathrm{O}(q^2)$ *and* $\varepsilon^* = \sigma^2/2^{n-1}$ *where the simulator* S *is described in Algorithm 5.3.*

6 Resistance of FWP Against Some Recent Attacks

One of the most significant works in hash function cryptanalysis in recent times is the discovery of the multicollision attack on the Merkle-Damgärd mode [9]. Using similar technique of multicollision attack, Kelsey and Schneier devised another very influential attack that recovered 2nd preimage with work lower than the brute-force when long messages were used in the Merkle-Damgärd mode. These two attacks do not work on the FWP mode. Any variants of these types of attacks do not seem to work too on the FWP transform. The above two attacks crucially rely on the intermediate collisions on n-bit chaining values which cannot be adjusted by message modification. The FWP mode has $2n$-bit chaining value which also cannot be adjusted by message modification. Therefore, the complexity of such attacks on the FWP mode appears to be no less than the brute-force. The same argument applies to the FWP's resistance to Herding attack [10] too. In the full version of the paper we shall provide further evidence why the FWP should be able to resist all variants of the above attacks.

6.1 Comparison of the FWP with Other Modes

The highlight of the FWP mode is that the compression function takes n bits of input but produces $2n$ bits of output. With the emergence of new types of attacks on the Merkle-Damgärd mode (see Sect. 6), it has been found necessary that the compression function output should be at least $2n$ bits to generate n bits of hash output. This type of constructions is known as the wide pipe mode propounded by Lucks [13] (see Fig. 1 (c)). Many modern hash functions use this type of mode [8] to defend against multi-collision type attacks. The main problem with that mode is that the $2n$ bits of chaining value, which is fed into the next compression function, reduce the bandwidth of the message-block and, thereby, impedes the speed of the hash function. To skirt this difficulty the Sponge construction with $2n$ bits of compression function output has been proposed [2] (see Fig. 1(d)). Unfortunately this construction collapses as easily as Merkle-Damgärd mode against all the attacks of Sect. 6. Another competing proposal is the HAIFA [4] mode. The HAIFA mode can be viewed as a special Merkle-Damgärd mode with an additional counter injected into each compression function call. This extra counter is very useful to thwart that attacks described in [11, 10]. However, the price to pay is the reduction of bandwidth for message in each compression function call, resulting in performance penalty. In addition, the HAIFA mode is still as weak against Joux's multi-collision attack as the old Merkle-Damgärd mode is. Pitted against all these modern modes of operation, the FWP mode, according to the definition and the security analyses in the previous sections, clearly outperforms all of them in terms of both security and performance.

7 Conclusion and Open Problems

This paper proposes a new sequential mode of operation, known as FWP, to hash messages of arbitrary length. The mode is *collision-resistance-preserving*, *preimage-resistance-preserving* and *indifferentiable* from a random oracle up to $2^{n/2}$ compression function invocations. The mode is also shown to be more efficient than the *Wide pipe* mode. No known attacks seem to be applicable in this mode. It may be possible to stretch the *indifferentiable* security bound of the mode beyond the birthday barrier of $2^{n/2}$. We leave this as an open problem.

References

[1] Mihir Bellare and Thomas Ristenpart. Multi-Property-Preserving Hash Domain Extension and the EMD Transform. In Xuejia Lai and Kefei Chen, editors, *ASIACRYPT 2006*, volume 4284 of *Lecture Notes in Computer Science*, pages 299–314. Springer, 2006. (Cited on pages 2 and 5.)

[2] 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. (Cited on pages 2 and 15.)

[3] Rishiraj Bhattacharya, Avradip Mandal, and Mridul Nandi. Indifferentiability Characterization of Hash Functions and Optimal Bounds of Popular Domain Extensions. Indocrypt 2009. (Cited on page 13.)

[4] Eli Biham and Orr Dunkelman. A framework for iterative hash functions – HAIFA. Second NIST Cryptographic Hash Workshop, 2006, 2006. (Cited on pages 2 and 15.)

[5] 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. (Cited on pages 16 and 17.)

[6] Jean-Sébastien Coron, Yevgeniy Dodis, Cécile Malinaud, and Prashant Puniya. Merkle-Damgård Revisited: How to Construct a Hash Function. In Victor Shoup, editor, *CRYPTO 2005*, volume 3621 of *Lecture Notes in Computer Science*, pages 430–448. Springer, 2005. (Cited on pages 2 and 6.)

[7] Ivan Damgård. A Design Principle for Hash Functions. In Brassard [5], pages 416–427. (Cited on page 1.)

[8] R. Rivest *et al.* The md6 hash function. (Cited on page 15.)

[9] Antoine Joux. Multicollisions in Iterated Hash Functions: Application to Cascaded Constructions. In *CRYPTO 2004*, pages 306–316, 2004. (Cited on pages 1, 2 and 15.)

[10] 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. (Cited on pages 2 and 15.)

[11] John Kelsey and Bruce Schneier. Second Preimages on n-Bit Hash Functions for Much Less than 2^{n} Work. In Ronald Cramer, editor, *EUROCRYPT 2005*, volume 3494 of *Lecture Notes in Computer Science*, pages 474–490. Springer, 2005. (Cited on pages 2 and 15.)

[12] A. Klimov and A. Shamir. New Cryptographic Primitives Based on Multiword T-Functions. In B. Roy and W. Meier, editors, *Fast Software Encryption 2004*, Lecture Notes in Computer Science, pages 1–15. Springer, 2004. (Cited on page 2.)

[13] 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. (Cited on pages 2 and 15.)

[14] Ueli M. Maurer, Renato Renner, and Clemens Holenstein. Indifferentiability, impossibility results on reductions, and applications to the random oracle methodology. In Moni Naor, editor, *TCC*, volume 2951 of *Lecture Notes in Computer Science*, pages 21–39. Springer, 2004. (Cited on pages 2, 5 and 6.)

[15] Ralph C. Merkle. One Way Hash Functions and DES. In Brassard [5], pages 428–446. (Cited on page 1.)

[16] NIST. Secure hash standard. In *Federal Information Processing Standard, FIPS-180*, 1993. (Cited on page 1.)

[17] NIST. Secure hash standard. In *Federal Information Processing Standard, FIPS 180-1*, April 1995. (Cited on page 1.)

[18] R. Rivest. The MD4 message-digest algorithm. In A. J. Menezes and S. Vanstone, editors, *CRYPTO*, volume 537 of *Lecture Notes in Computer Science*, pages 303–311. Springer, 1990. (Cited on page 1.)

[19] R. Rivest. The MD5 message-digest algorithm. In *IETF RFC 1321*, 1992. (Cited on page 1.)

[20] David Wagner. A Generalized Birthday Problem. In Moti Yung, editor, *CRYPTO*, volume 2442 of *Lecture Notes in Computer Science*, pages 288–303. Springer, 2002. (Cited on page 4.)