The Hash Function Family LAKE

Jean-Philippe Aumasson[1]*, Willi Meier[1], and Raphael C.-W. Phan[2]**

[1] FHNW, 5210 Windisch, Switzerland
[2] Electronic & Electrical Engineering, Loughborough University, LE11 3TU, United Kingdom

Abstract. This paper advocates a new hash function family based on the HAIFA framework, inheriting built-in randomized hashing and higher security guarantees than the Merkle-Damgård construction against generic attacks. The family has as its special design features: a nested feed-forward mechanism and an internal wide-pipe construction within the compression function. As examples, we give two proposed instances that compute 256- and 512-bit digests, with a 8- and 10-round compression function respectively.

Keywords: Hash function, HAIFA, Randomized hashing, Salt, Wide-pipe.

1 Introduction

Why do we need another hash function? Aside from the explicit aim of the U.S. Institute of Standards and Technology (NIST) to revise its standards [29,30], motivations lie in the present status of hash functions: among the proposals of recent years, many have been broken (including "proven secure" ones), or show impractical parameters and/or performance compared to the SHA-2 functions, despite containing interesting design ideas[3]. For example all the hash functions proposed at FSE in the last five years [17–19] are now broken [25, 34, 35], except for one [33] based on SHA-256. We even see recent works [27] breaking old designs that had until now been assumed secure due to absence of attacks. It seems necessary to learn from these attacks and propose new hash functions that are more resistant to known cryptanalysis methods, especially against differential-based attacks, which lie at the heart of major hash function breaks. These design proposals would also hopefully contribute to the discovery of new ways to attack hash functions, contributing to NIST's SHA3 development effort.

This paper introduces the hash function family LAKE along with two particular instances aimed at suiting a wide variety of cryptographic usages as well as present and future API's. We adopted the extended Merkle-Damgård framework HAIFA [8] and include the following features in our design:

- **Built-in salted hashing**: to avoid extra code for applications requiring a salt (be it random, a nonce, etc.), and to encourage the use of randomized hashing.
- **Software-oriented**: we target efficiency in software, although efficient hardware implementations are not precluded.
- **Direct security**: the security is not conditioned on any external hardness assumption.
- **High speed**: with significantly better performance than the SHA-2 functions on all our machines.
- **Flexibility**: with variable number of rounds and digest length.

* Supported by the Swiss National Science Foundation under project number 113329.

** Work done while the author was with the Security & Cryptography Lab (LASEC), Ecole Polytechnique Fédérale de Lausanne (EPFL), Switzerland.

[3] Note that the ISO/IEC standards Whirlpool [3] and RIPEMD-160 [13] are not broken yet.

Road Map. First we give a complete specification of LAKE (§2), then we explain our design choices (§3), present performance (§4), and study security of the proposed instances (§5). Appendices include lists of constants, and test values.

2 Specification

This section gives a bottom-up specification of the LAKE family, and the definition of the instances LAKE-256 and LAKE-512 ("LAKE" designates both the general structure and the family of hash functions built upon it, while instances have parametrized names). We'll meet the following symbols throughout the paper (length unit is the bit).

w	Length of a word	$\tilde{M}$	Padded message
n	Length of the chaining variable	$\tilde{M}^t$	t-th (padded) message block
m	Length of the message block	$\tilde{M}_i^t$	i-th word of $\tilde{M}^t$
s	Length of the salt	N	Number of blocks of $\tilde{M}$
d	Length of the digest	S	Salt
b	Length of the block index	S_i	i-th word of the salt
r	Number of rounds	H^t	t-th chaining variable
t	Index of the current block	H_i^t	i-th word of H^t
t_i	i-th word of t	D	Message digest
M	Message to hash	IV	n-bit fixed initial value

We let lengths n, m, etc. be multiples of w. Hexadecimal numbers are written in typewriter style, and function or primitive labels in sans-serif font. The operator symbols $+$, $\oplus$, $\gg$, $\ggg$, $\vee$, $\wedge$ keep their standard definition. Last but not least, LAKE is defined in *unsigned little-endian* representation.

2.1 Building Blocks

LAKE's compression function compress is made up of three procedures: *initialization* (function saltstate), internal *round function* (function processmessage), and *finalization* (function feedforward). Fig. 1 represents the interaction between these functions.

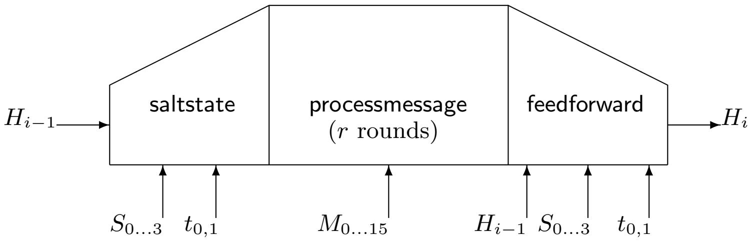

Fig. 1. The structure of LAKE's compression function: the chaining variable H_{i-1} is transformed into a local chaining variable twice as large, which undergoes message-dependent modification, before being shrunk to define H_i.

The saltstate Function. This mixes the global chaining variable H with the salt S and the block index t, writing its output into the buffer L, twice as large as H. In addition, saltstate uses 16 constants $C_0, \ldots, C_{15}$, and a function g that maps a $4w$-bit string to a w-bit string. L plays the role of the chain variable—in a "wide-pipe" fashion—, while g can be seen as an internal compression function, and the constants are used to simulate different functions. In the following algorithm, word indexes are taken modulo the number of w-bit words in the array. For example, in the second "for" loop of saltstate's algorithm, i ranges from $2 \equiv 10 \pmod 8$ to $7 \equiv 15 \pmod 8$ for H_i, and over $2, 3, 0, 1, \ldots, 2, 3$ for S_i (see also Fig. 2).

```
saltstate
input $H = H_0 \| \ldots \| H_7, \quad S = S_0 \| \ldots \| S_3, \quad t = t_0 \| t_1$

1. for $i = 0, \ldots, 7$  do
       $L_i \leftarrow H_i$
2. $L_8 \leftarrow \mathsf{g}(H_0, S_0 \oplus t_0, C_8, 0)$
3. $L_9 \leftarrow \mathsf{g}(H_1, S_1 \oplus t_1, C_9, 0)$
4. for $i = 10, \ldots, 15$  do
       $L_i \leftarrow \mathsf{g}(H_i, S_i, C_i, 0)$

output $L = L_0 \| \ldots \| L_{15}$
```

The first eight words in the buffer L allow H to pass through saltstate unchanged for use in later feedforwarding, while the last eight words ensure dependence on the salt and the block index. Clearly, the function is not surjective, but will be injective for a well-chosen g. These local properties do not raise any security issues (see §5 for more discussion).

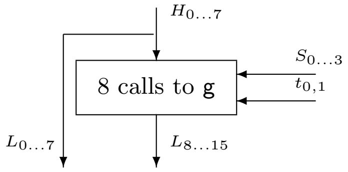

Fig. 2. The saltstate function.

The processmessage Function. This is the bulk of LAKE's round function. It incorporates the current message block M within the current internal chaining variable L, with respect to a permutation σ of the message words. It employs a local m-bit buffer F for local feedforward, and internal compression functions f and g, both mapping a $4w$-bit string to a w-bit string. In the algorithm below, indexes are reduced modulo 16, i.e. $L_{-1} = L_{15}$, $L_{16} = L_0$.

```
processmessage
input $L = L_0\| \dots \|L_{15}$,  $M = M_0\| \dots \|M_{15}$,  $\sigma$

1. $F \leftarrow L$
2. for $i = 0, \dots, 15$ do
       $L_i \leftarrow \mathsf{f}(L_{i-1}, L_i, M_{\sigma(i)}, C_i)$
3. for $i = 0, \dots, 15$ do
       $L_i \leftarrow \mathsf{g}(L_{i-1}, L_i, F_i, L_{i+1})$

output $L = L_0\| \dots \|L_{15}$
```

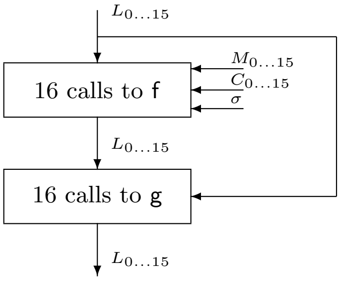

Fig. 3. The processmessage function.

The feedforward Function. This compresses the initial global chaining variable H, the salt S, the block index t and the hitherto processed internal chaining variable L. It outputes the next chaining variable. In the algorithm indexes are reduced modulo 4 for S (see also Fig. 4).

```
feedforward
input $L = L_0\| \dots \|L_{15}$,  $H = H_0\| \dots \|H_7$,  $S = S_0\| \dots \|S_3$,  $t = t_0\|t_1$

1. $H_0 \leftarrow \mathsf{f}(L_0, L_8, S_0 \oplus t_0, H_0)$
2. $H_1 \leftarrow \mathsf{f}(L_1, L_9, S_1 \oplus t_1, H_1)$
3. for $i = 2, \dots, 7$ do
       $H_i \leftarrow \mathsf{f}(L_i, L_{i+8}, S_i, H_i)$

output $H = H_0\| \dots \|H_7$
```

The compress Function. This is the *compression function* of LAKE. It computes the next chaining variable H^{t+1} from the current H^t, the current message block M^t, the salt S and the current block index t. The number of rounds r and the permutations $(\sigma_i)_{0 \le i < r}$ are parameters to be set later.

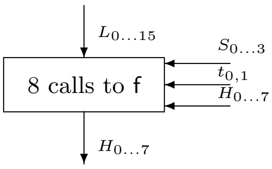

Fig. 4. The feedforward function.

```
compress
input $H = H_0\|\ldots\|H_7$,  $M = M_0\|\ldots\|M_{15}$,  $S = S_0\|\ldots\|S_3$,  $t = t_0\|t_1$

1. $L \leftarrow \mathsf{saltstate}(H, S, t)$
2. for $i = 0, \ldots, r-1$ do
       $L \leftarrow \mathsf{processmessage}(L, M, \sigma_i)$
3. $H \leftarrow \mathsf{feedforward}(L, H, S, t)$

output $H$
```

Apart from its arguments, compress requires 32 memory words for L and F. Here, L acts as an internal chaining variable, twice as long as H, finally shrunk to output the next chaining variable, as in the "wide-pipe" construction [23, 24]. The goal of this approach is to make local collisions difficult to find—if not impossible. Observe that the current message block M is input r times to (and thus input to r different points of) processmessage; meanwhile the salt S, the current block index t, and the chaining variable H are feedforwarded to the last stage of compress, thus in fact these inputs are injected into two different points of compress.

2.2 The LAKE Structure

The LAKE structure consists of the sequence: *initialization* (functions pad and init), *iteration* of compress, and *finalization* (function out). We start this section with a description of the padding rule, inherited from HAIFA.

Padding. A message M is padded by concatenating a '1' bit followed by sufficient number of '0' bits, then the b-bit message length, and the digest length d, such that a padded message $\tilde{M}$ is exactly km bits, for a minimal integer k.

Initialization. The effective initial chaining variable H^0 is computed by the function init on input an n-bit initial value IV and a length d of the output hash value:

$$H^0 \leftarrow \mathsf{init}(IV, d) = \mathsf{compress}(IV, d, 0, 0).$$

For words of reasonable length d is written in M_0, and all other M_i's are null. In practice H^0 should be precomputed, unless variable digest length is necessary.

Finalization. The function out simply truncates the final chaining variable H^N to its d first bits, to return the final hash output digest D.

Overall Hashing. A LAKE hash function take as input a message and a salt, and is parametrized by an initial value IV, a number of rounds r, a sequence of permutations $(\sigma_i)_{0\leq i<r}$, the word size w, and subsequently bit-lengths n, m, d, s, b.

```
LAKE
input $M = M^0 \| \dots \| M^{N-1}$,   $S = S_0 \| \dots \| S_3$

1. $\tilde{M} \leftarrow \mathsf{pad}(M)$
2. $H^0 \leftarrow \mathsf{init}(IV, d)$
3. for $t = 0, \dots, N-1$ do
       $H^{t+1} \leftarrow \mathsf{compress}(H^t, \tilde{M}^t, S, t)$
4. $D \leftarrow \mathsf{out}(H^N)$

output $D$
```

2.3 Instances

We introduce LAKE-256 and LAKE-512, respectively suited for 32- and 64-bit words:

LAKE-256 has parameters

$$\begin{array}{lll} n\ = 256 \text{ (chaining variable)} & d = 256 \text{ (digest)} & r = 8 \ \ \text{(rounds)} \\ m = 512 \text{ (message block)} & s = 128 \text{ (salt)} & b = 64 \text{ (block index)} \end{array}$$

Its IV and constants $C_0, \dots, C_{15}$ are extracted from π (see Appendix A), and permutations of the set $\{0, \dots, 15\}$ are defined as in MD5 by

$$\begin{array}{lll} i \equiv 0 \pmod 4 \Rightarrow \sigma_i(j) = j & \\ i \equiv 1 \pmod 4 \Rightarrow \sigma_i(j) = 5j + 1 & \pmod{16} \\ i \equiv 2 \pmod 4 \Rightarrow \sigma_i(j) = 3j + 5 & \pmod{16} \\ i \equiv 3 \pmod 4 \Rightarrow \sigma_i(j) = 7j & \pmod{16} \end{array}$$

The internal compression functions are[4]

$$\begin{aligned} \mathsf{f}(a,b,c,d) &= \big[a + (b \vee C_0)\big] + \Big(\big[c + (a \wedge C_1)\big] \ggg 7\Big) + \Big(\big[b + (c \oplus d)\big] \ggg 13\Big) \\ \mathsf{g}(a,b,c,d) &= \big[(a+b) \ggg 1\big] \oplus (c+d). \end{aligned}$$

LAKE-512 has parameters

$$\begin{array}{lll} n\ = 512 \ \ \text{(chaining variable)} & d = 512 \text{ (digest)} & r = 10 \ \ \text{(rounds)} \\ m = 1024 \text{ (message block)} & s = 256 \text{ (salt)} & b = 128 \text{ (block index)} \end{array}$$

Constant values are given in Appendix A, permutations are the same as for LAKE-256, and internal compression functions are

$$\begin{aligned} \mathsf{f}(a,b,c,d) &= \big[a + (b \vee C_0)\big] + \Big(\big[c + (a \wedge C_1)\big] \ggg 17\Big) + \Big(\big[b + (c \oplus d)\big] \ggg 23\Big) \\ \mathsf{g}(a,b,c,d) &= \big[(a+b) \ggg 1\big] \oplus (c+d). \end{aligned}$$

[4] It was observed by F. Mendel, C. Rechberger, and M. Schläffer [26] that the non-invertibility of f can be exploited to find collisions for a reduced version with 4 rounds (instead of 8) faster than with a birthday attack (this was independently suggested by S. Lucks at FSE 2008). These authors mention that choosing e.g. $\mathsf{f}(a,b,c,d) = [a + (b \vee C_0)] + ([d + (a \wedge C_1)] \ggg 7) + ([b + (c \oplus d)] \ggg 13)$ makes this attack impossible. It is however unclear whether their suggestion impacts the security of the original version with all 8 rounds.

Other Instances. Instances with digest length $0 < d \leq 256$ take LAKE-256 parameters, and instances with $256 < d \leq 512$ take LAKE-512 parameters. Since the effective initial value H^0 depends on d, this will be distinct for each choice of d.

3 Design

Design rationale is given top-down, from the operating mode to the wordwise operators. Apart from the obvious concerns of security and speed guiding principles include:

- **Withstand differential attacks**: no high-probability differential path should be exploitable, including techniques based on impossible or truncated differentials.
- **Prevent side-channel leakage**: time and memory consumption as well as operations should be input-independent, to avoid weaknesses in keyed modes.
- **Facilitate implementation**: instances proposed should allow compact implementations, be the less processor-specific as possible, use simple operators.
- **Facilitate analysis**: we use a small number of building blocks, of reasonable complexity, and provide flexible instances, in a clear and concise specification.

3.1 HAIFA as the Operating Mode

Some properties of the classical MD mode and the need for salted hashing (not explicitly handled by the previous constructions) motivated the design of the HAsh Iterative FrAmework (HAIFA) [8]. Its main novelties are the explicit input of a salt and the number of bits hashed so far to the compression function, the computation of the initial value depending on the digest length, and the padding rule. Consequently,

- generic attacks for finding "one-of-many" preimages and second-preimages with k targets requires 2^d trials for HAIFA against $2^d/k$ for MD,
- online "herding" time-memory trade-off for finding preimages with memory 2^t require $2^{d/2+t/2+s}$ trials for HAIFA against $2^{d/2+t/2}$ for MD,
- HAIFA captures the MD, enveloped MD [5], RMC [1], ROX [2], and Wide-pipe [23, 24] operating modes,
- it explicitly handles randomized hashing.

Other operating modes could have been chosen, however we believe that HAIFA offers more advantages, in addition to a simple and familiar framework for cryptanalysts and implementers since it builds on the well known MD mode.

On Salted Hashing. To the best of our knowledge, LAKE is the first concrete hash construction to include built-in salting. Although this is not a strict requirement for randomized hashing (see e.g. the RMX transforms of Halevi and Krawczyk [15]), it has the advantage of requiring no additional programming, so that the hash function can be directly used as a black box fed with a message and a salt. The main application of randomized hashing is digital signatures, enhancing security by reducing the security requirement of the underlying hash function from collision-resistance to second-preimage-like notions [15]. More generally, a salt may find applications in protocols requiring hash functions families, as well as future protocols may take advantage of it, be it public or secret, random or a counter. When a salt is not needed, the null value can be used.

A disadvantage of salted hashing is that extra-data might have to be sent, while disadvantages of built-in salt are that it might facilitate certain attacks, and potentially increases the complexity of the algorithm. On the other hand, it encourages the use of randomized hashing, and prevents from weak home-brewed construction.

3.2 Building Blocks

The structure of compress is similar in spirit to the one of the overall hash function: initialization, chained rounds, and finalization. Essential differences are that the round function here allows no collision for a fixed chaining variable, and that the same message is used at each iteration (up to a permutation of words). The goal of the internal wide-pipe is indeed to have an (almost) injective function built as a repetition of processmessage; the goal is to make local collisions unlikely for fixed chaining variable, salt, and counter. Also note that, contrary to the SHA functions, LAKE-256 does not admit easily found fixed-points.

For designing compress we create two levels of interdependence: fast diffusion can be seen as a propagation of *spatial interdependence* across words at any intermediate state within the hash function, and complicates attacks including those that exploit high-probability differentials. Analogously, the feedforward mechanism allows injected values to influence two different intermediate states at different times during the processing, which achieves *temporal interdependence*. Although wordwise diffusion as a spatial interdependence technique is widely known in cryptographic literature to increase resistance to common attacks, the notion of feedforwarding as a form of achieving temporal interdependence is less treated. In fact, this latter serves to complicate the perturb-and-correct strategy used in many hash function attacks that exploit inputs with well chosen differences. The central building block processmessage achieves both spatial and temporal interdependence by making use of multiple-blockwise chaining [4] and feedforwarding respectively. Spatial and temporal interdependencies are achieved in a similar way within feedforward and the structure of the compression function compress. We further comment on the three internal modules of compress hereafter.

We chose a round-dependent permutation for the message input, with same permutations as MD5; note that, though MD5 is broken, it is essentially due to the relative simplicity of its round function, rather than to the message input strategy. An alternative would be to use message expansion, as used in a fully XOR-linear fashion in SHA-0/1, and non-linearly in SHA-2. The main argument for recursive message expansion is that it simulates a complicated function, and the non-surjectivity makes collisions of random expanded messages useless. However, it may increase memory usage, and other strategies can be used to have a complex mixing.

3.3 Core Functions f and g

Each of these functions is called 136 times in an 8-round compress. We opted for a high-number of calls to simple functions, rather than a few calls to some complicated procedures, mainly because it simplifies analysis and implementation, and reduces the amount of code.

The role of f is to provide a large amount of mixing, to break linearity and diffuse changes across words. We considered various combinations of wordwise operators and our final choice was selected for its high ratio diffusion over speed, its ability to increase quickly the algebraic degree, and its simplicity. The much simpler g only aims at making each input influence the internal state, within a progressive diffusion of changes via addition carries and 1-bit rotations in a non-linear fashion. When used as arguments, constants C_i simulate distinct functions and reduces self-similarity.

3.4 Wordwise Operators

We chose a combination of standard constant-time word operators, known to be complementary to achieve cryptographic strength: integer addition and XOR diffuse changes locally, while logical operators AND and OR increase non-linearity—over GF(2) and GF(2^{32}). Though the operators AND and OR are in no way mandatory, the use of the sole triplet $(+, \oplus, \ggg)$ can be risky, as suggested by the existence of high-probability differentials in the stream ciphers Phelix and Salsa20/8, the block cipher TEA, or in the hash function FORK-256 [7, 17, 39, 40]. Finally, rotation provides fast diffusion within the words, with a choice of data-independent distances—in order to avoid side-channel leakage, reduce the control of the attacker over the operations, and reduce complexity of the algorithm. Rotation counts of f were chosen so as to avoid byte alignments, and diffuse changes to any word offset as fast as possible; as observed by the authors of Twofish [37], one-bit rotation (as used in g) saves time over smartcards processors, compared to multi-bit rotation. We avoid integer multiplication essentially for performance reasons and the risk of timing leakage.

3.5 Parameters

We propose instances whose input and output have lengths similar to previous and current standards, to suit present and future API's, and minimize implementers' work. The salt length was chosen to be sufficient for randomized hashing, and to suit HAIFA's requirements (for which the salt should be at least half as large as the digest).

After intensive security analysis, we believe that eight rounds for LAKE-256 are sufficient for actual security, and as a security margin to counter future attacks (in comparison, MD5 and SHA-256 have four rounds, SHA-1 and SHA-512 have five rounds). We add two rounds for LAKE-512 whose larger state delays full diffusion.

4 Performance

4.1 Algorithmic Complexity

We consider here the algorithm independently from any specific implementation or parallelism issues, and study time and space complexities. However, from this abstract evaluation one cannot directly infer statements on the actual speed of the algorithms when implemented, which depends on a multitude of other factors (see speed benchmarks in §4.2).

Table 1 presents on its leftmost part the number of arithmetic operations for LAKE-256, LAKE-512, and their components, comparing with SHA-256 and SHA-512 (refering to [29]): our functions count slightly less operations than the SHA-2 equivalents, with significantly less XORs and more integer additions. The larger amount of rotations in SHA-2 functions increases the difference of operation counts on processors simulating a rotation with two shifts (as the Itanium and UltraSPARC). The rightmost part of Table 1 compares storage requirements.

Table 1. Algorithmic complexities and memory requirements (in bytes).

Function	Operations							Memory	
	Total	$+$	$\oplus$	$\ggg$	$\gg$	$\vee$	$\wedge$	ROM	RAM
f	10	5	1	2	0	1	1	-	-
g	4	2	1	1	0	0	0	-	-
saltstate	34	16	10	8	0	0	0	-	-
processmessage	224	112	32	48	0	16	16	-	-
feedforward	82	40	10	16	0	8	8	-	-
LAKE-256	1908	952	276	408	0	136	136	64	128
SHA-256	2232	600	640	576	96	0	320	256	64
LAKE-512	2356	1176	340	504	0	168	168	128	256
SHA-512	2632	712	752	672	96	0	400	512	128

4.2 Implementation

Implementation on 32- and 64-bit architectures should pose no problem, since we only use standard wordwise operators. On 8- and 16-bit architectures (e.g. smartcards) word operations have to be decomposed; rotations translate less simply than addition and XOR, but remain easily implementable. Choosing multiples of 8 as rotation counts would have improved performances on 8- and 16-bit processors, but reduced the quality of diffusion. We do not preclude hardware implementation, since our operators are consistently simple and fast.

Speed Benchmarks. It is, alas, rather difficult to make a complete and comprehensive relative study of hash functions: security evaluation requires intensive cryptanalysis effort (even for "proven secure" designs), and performances comparison cannot be fair when reference source codes are not published, or do not have a same degree of optimization, are more or less processor-dependent, etc., not to mention the issue of hardware benchmarks. The simplest solution is then to compare with the reference SHA-2 family, since other proposals would also be compared to these functions.

We compare LAKE-256 with SHA-256 using portable C implementations: respectively our reference code (available upon request) and the version in XySSL [12]. These codes have roughly the same level of optimization, and we used exactly the same source code for all processors. Cycles counts are measured using the RDTSC assembly instruction through the processor-specific cpucycles library [6], on machines running a Linux kernel 2.6.19 with Gentoo distributions; sources are compiled with `gcc 4.1.2` with full optimization flags (`-O3`) and processor-specific settings (e.g. `-march=pentium4`). For each machine, Table 2 shows the *median cycles count* measured among 1000 successive calls to the compression function with random inputs, along with the cycles-per-byte cost. This measurement has a relatively high variant, so we give rounded values for clarity, that seems sufficient for a raw comparison of performance.

LAKE-256 significantly outperforms SHA-256 on our test machines, particularly on our Athlon XP and Pentium D, while the difference of cycles on the other machines roughly matches the difference of arithmetic operations (see §4.1); this suggests that LAKE-256 takes a particular advantage of some feature of the former processors (possibly thread-level parallelism). These results should be interpreted carefully, and performance on other architectures as well as the optimization potential remain to be studied.

Table 2. Cycle counts for the compression function (and corresponding cycles-per-byte cost).

Name	CPU Frequency	L2 cache	Function LAKE-256	SHA-256
Athlon	800 MHz	256 Kb	2700 (42)	3000 (50)
Athlon XP	1830 MHz	512 Kb	2400 (38)	4500 (70)
Pentium 4	1500 MHz	256 Kb	3600 (56)	4050 (63)
Pentium 4	2400 MHz	512 Kb	3300 (52)	3900 (61)
Pentium D	2×3010 MHz	2048 Kb	2600 (41)	4500 (70)

Parallelism. How can LAKE-256 be parallelized? First consider the "medium grain" level: In saltstate, the computation of the L_i's can be parallelized into sixteen branches, since there is no diffusion across word boundaries—the concurrent access to H might however be an obstacle. Due to its large amount of flow dependence, the main function processmessage is not parallelizable, unlike feedforward, which can be split into eight branches. At a finer level, the three internal expressions of f can be computed in parallel (by copying two variables), as well as the two additions of g. This can benefit to the three components.

5 Security

5.1 Introduction

Definitions. For LAKE hash functions, the preimage problem (resp. second preimage) takes as parameter a random digest y (resp. random message and salt of digest y), and the challenge is to find a (distinct) pair message and salt mapping to y. Target second preimage is similar to second preimage except that the two salts must be identical. The collision problem is to find two pairs message and salt with identical digest, and we call a collision *synchronized* if the two salts are identical. The generic (brute force) attack solves preimage with probability ε within $2^{d+\log_2 \varepsilon}$ calls to the hash function. Hellman's time(T)-memory(M) trade-off is $TM^2 = 2^{2n}$ [16]. Idem for (target) second preimage. Against collision, the generic birthday attack requires $\sqrt{-2\log(1-\varepsilon)}\cdot 2^{d/2}$ calls to the hash function to succeed with probability ε, with negligible memory requirement (thanks to smart variants of Floyd's cycle-finding technique [36, 38]). For evaluating the cryptographic quality of the distributions induced by our hash functions, we suggest to consider the definitions of *pseudo-randomness* and *unpredictability* given by Naor and Reingold [28] for function distributions, which apply as well for salt-indexed families derived from a LAKE instance (in this case the function of the family take as sole input a message).

Conjectures. For all the instances proposed, no method should solve preimage, second preimage, or collision faster than the generic one for any parameters of the problem. We also conjecture pseudo-randomness and unpredictability for families indexed by the s-bit salt. In addition, relaxed problems as pseudo- or near-collision should also be hard (note that our claims concern the *full functions*, not the building blocks individually). On the other hand we expect variants with less than three rounds to be relatively weak—though we have no evidence of it yet.

5.2 One-Wayness and Collision Resistance

One-wayness is achieved mainly thanks to the feedforward operations (in processmessage and feedforward), and to the redundancy in the initial local chaining variable L (in saltstate).

Arguments for collision resistance are given by the structure of processmessage: recall from §3.2 that the local wide-pipe strategy of processmessage makes it injective for a fixed L; we can expect the r-round processmessage to be collision-free as well, or at least have a negligible number of collisions. Synchronized collisions on compress can thus only occur at the ultimate step, feedforward, when the 512-bit state is compressed to a 256-bit chaining variable (for LAKE-256 parameters). Therefore, the objective of the repeated processmessage is to make hard the search for message pairs (M, M') such that

$$\mathsf{feedforward}(\mathsf{processmessage}(L, M, \sigma), H, S, t) = \mathsf{feedforward}(\mathsf{processmessage}(L, M', \sigma), H, S, t),$$

with $L = \mathsf{saltstate}(H, S, t)$. Note that, if synchronized collisions over compress with similar (H, S, t) are hard to find, then the corresponding LAKE instance is target collision resistant. This is because for any collision over the full hash function with similar salts, at least one collision over compress with identical counter exists.

On the other hand, it is easy to find 1-round collisions with (chosen) distinct salts, and identical H: it suffices to take a random message M, and adapt a second one M' to correct changes in the last eight words of the initial L. However, the collision does not persist to subsequent rounds, and seems to have no consequence on the overall security of compress.

5.3 Algebraic Attacks

Traditional algebraic attacks aim at giving an input/output relationship in terms of multivariate equations, then exploiting this system (ideally solving it) to recover secret information. Due to its rather nested structure, LAKE is unlikely to be vulnerable to such attacks: addition carries and chained computation of L ensure an algebraic normal form (ANF) for each Boolean component of maximal degree after one round of processmessage.

Recently, two works aimed at detecting non-uniform randomness in the algebraic structure of several cryptographic primitives [14, 31, 32]. Their basic idea is to compute all or part of the ANF of some implicit Boolean function, mapping part of the input to part of the output, then applying statistical tests on the distribution of the monomials of the ANF; reference [32] notably claims to distinguish SHA-2 functions from random using so-called Defectoscopy, claiming that *"at least 8 full cycles are required for it to be secure instead of the proposed 4-5"*, *idem* for MD5 and SHA-0/1—unfortunately, the description of the tests and the methodology used in [32] is not precise enough to reproduce the experiments. In [14], comparable tests are used to build distinguishers for the stream ciphers DECIM, Grain, Lex, and Trivium. Against such methods, we applied the following countermeasures:

- *Intensive feedforward*: each round incorporates a complex feedforward operation, such temporal dependency providing highly non-linear relations.
- *Many additions*: compared to SHA-256, we use a large amount of integer additions, and a few XORs (see Table 1), which increases non-linearity through the carries.
- *Large state*: the output of compress is extracted from a large local state combined with a non-linear dependence on the initial H in feedforward (unlike in SHA-2 functions).

We ran a few experiments with the "d-monomial" and "maximum-degree monomial" tests described in [14], and also used in [32]; for input windows of 8 to 20 bits of the salt or of the message, and each of the 256 output bits. Significant deviations where observed for up to two rounds. This bound might be slightly increased by using refined experiments, as apparently employed, but not detailed, in [32].

5.4 Differential Attacks

The essential idea of differential attacks on hash functions [10], as used to break MD5 and SHA-0/1, is to exploit a high-probability input/output differential over some component of the hash function, e.g. under the form of a "perturb-and-correct" strategy for the latter functions, exploiting high-probability linear/non-linear characteristics. A common property of those functions, as well as to the SHA-2 functions, is indeed the behavior of the compression function as a shift register: at each step, a "first" word is updated depending of a message word input, while the other words are shifted. Hence "corrected words" can spread along the state, as explained in the Chabaud-Joux attack on SHA-0 [11]. Moreover, their relatively simple step function allowed to find several high-probability characteristics (see e.g. [10])—the SHA-2 functions made this much more difficult, though have more or less the same structure as their ancestors, and keep a similar number of rounds.

In the design of LAKE-256 and LAKE-512, we applied the following countermeasures against differential attacks:

- *High number of steps*: with respectively 128 and 160 message word inputs in LAKE-256 and LAKE-512, against 64 for MD5 and SHA-256, and 80 for SHA-512 and SHA-0/1. In particular, the function f called 136 and 168 times in compress makes the exploit of linear approximations highly implausible.
- *Nested feedforward*: the r message-dependent internal feedforward operations aim at strengthening the function against differential paths.
- *Internal wide-pipe*: this makes internal collisions unlikely, and the final compression of L to H makes differences in the output much harder to predict.
- *No "shift register"*: in the round function, *all* state words are updated in chain with dependence on a message word, and then undergo a message-independent post-treatment, making any "correction" impossible.
- *Use all operators*: as observed in §3.4, a small set of operations often facilitates differential analysis.

Note that the foregoing features, except the increased number of rounds, do not require extra computation or memory, unlike the use of a recursive message expansion, or of S-boxes.

We can sketch a simple method for finding low-weight 1-round differentials in compress: choose an input difference Δ changing M_{14} and M_{15} such that after the first loop in processmessage, only L_{14} is modified. Consequently, after the second loop changes will occur only in L_{13}, L_{14} and L_{15}, that is, a difference of weight at least 3. However, such low-weight output-difference will persist no further. We discovered no high-probability differential, but a more careful analysis is required.

5.5 Empirical Tests

For completeness, we report some experiments assessing the minimal requirements for a hash function. Note that no statement about preimage or collision resistance should be derived from these results.

Diffusion. To illustrate the difference propagation in LAKE-256, we give visual examples of the diffusion provided by processmessage, after running saltstate with $H = IV$, $S = 0$, $t = 0$.

The avalanche effect is suggested by the high number of differences within only two rounds of processmessage. We consider various one-bit differences in random messages, as presented in Fig. 5: the first stripe represents the message difference, and the eight subsequent ones show the differences in the buffer L after each round of processmessage.

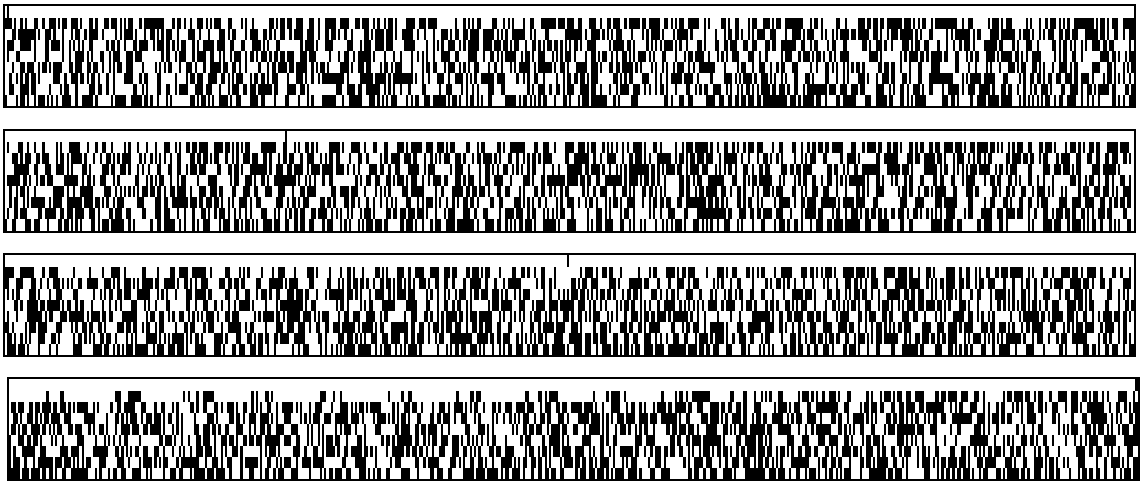

Fig. 5. Diffusion diagrams, for randomly chosen messages and a difference at 2nd, 128th, 256th, and 512th position.

The observation that the most-significant bits of the message diffuse less after one round can easily be explained by the algorithm of processmessage. This however has no consequence on the security *per se*, since after only two rounds no kind of regularity seems observable.

Acknowledgments

We would like to thank the reviewers of FSE 2008 who pointed out several issues to improve the paper, and F. Mendel, C. Rechberger, M. Schläffer, and S. Lucks for their analysis of LAKE .

References

1. Elena Andreeva, Gregory Neven, Bart Preneel, and Thomas Shrimpton. Seven-properties-preserving iterated hashing: The RMC construction. Technical Report STVL4-KUL15-RMC-1.0, ECRYPT, 2006.
2. Elena Andreeva, Gregory Neven, Bart Preneel, and Thomas Shrimpton. Seven-property-preserving iterated hashing: ROX. In Kurosawa [21], pages 130–146.
3. Paulo Barreto and Vincent Rijmen. The Whirlpool hashing function. First Open NESSIE Workshop, 2000.
4. Mihir Bellare, Oded Goldreich, and Shafi Goldwasser. Incremental cryptography: The case of hashing and signing. In Yvo Desmedt, editor, *CRYPTO*, volume 839 of *LNCS*, pages 216–233. Springer, 1994.
5. Mihir Bellare and Thomas Ristenpart. Multi-property-preserving hash domain extension and the EMD transform. In Lai and Chen [22], pages 299–314.
6. Daniel J. Bernstein. The cpucycles library. `http://ebats.cr.yp.to/cpucycles.html`.
7. Daniel J. Bernstein. Salsa20. Technical Report 2005/25, ECRYPT eSTREAM, 2005. Papers and code available at `http://cr.yp.to/snuffle.html`.
8. Eli Biham and Orr Dunkelman. A framework for iterative hash functions - HAIFA. Cryptology ePrint Archive, Report 2007/278, 2007. Previously presented at the second NIST Hash Function Workshop, 2006.

9. Alex Biryukov, editor. *Fast Software Encryption, 14th International Workshop, FSE 2007, Luxembourg, Luxembourg, March 26-28, 2007, Revised Selected Papers*, volume 4593 of *LNCS*. Springer, 2007.
10. Christophe De Cannière and Christian Rechberger. Finding SHA-1 characteristics: General results and applications. In Lai and Chen [22], pages 1–20.
11. Florent Chabaud and Antoine Joux. Differential collisions in SHA-0. In Krawczyk [20], pages 56–71.
12. Christophe Devine. XySSL. `http://xyssl.org/code/`.
13. Hans Dobbertin, Antoon Bosselaers, and Bart Preneel. RIPEMD-160: A strengthened version of RIPEMD. In Dieter Gollmann, editor, *Fast Software Encryption*, volume 1039 of *LNCS*, pages 71–82. Springer, 1996.
14. Hakan Englund, Thomas Johansson, and Meltem Sonmez Turan. A framework for chosen IV statistical analysis of stream ciphers. In *Special ECRYPT Workshop – Tools for Cryptanalysis*, 2007. Available at `http://www.impan.gov.pl/BC/Program/conferences/07Crypt-prg.html`.
15. Shai Halevi and Hugo Krawczyk. Strengthening digital signatures via randomized hashing. In Cynthia Dwork, editor, *CRYPTO*, volume 4117 of *LNCS*, pages 41–59. Springer, 2006.
16. Martin Hellman. A cryptanalytic time-memory tradeoff. *IEEE Transactions on Information Theory*, 26:401–406, 1980.
17. Deukjo Hong, Donghoon Chang, Jaechul Sung, Sangjin Lee, Seokhie Hong, Jaesang Lee, Dukjae Moon, and Sungtaek Chee. A new dedicated 256-bit hash function: FORK-256. In Matthew J. B. Robshaw, editor, *FSE*, volume 4047 of *LNCS*, pages 195–209. Springer, 2006.
18. Lars R. Knudsen. SMASH - a cryptographic hash function. In Henri Gilbert and Helena Handschuh, editors, *FSE*, volume 3557 of *LNCS*, pages 228–242. Springer, 2005.
19. Lars R. Knudsen, Christian Rechberger, and Søren S. Thomsen. The Grindahl hash functions. In Biryukov [9], pages 39–57. Paper and code available at `http://www.ramkilde.com/grindahl/`.
20. Hugo Krawczyk, editor. *Advances in Cryptology - CRYPTO'98, 18th Annual International Cryptology Conference, Santa Barbara, California, USA, August 23-27, 1998, Proceedings*, volume 1462 of *LNCS*. Springer, 1998.
21. Kaoru Kurosawa, editor. *Advances in Cryptology - ASIACRYPT 2007, 13th International Conference on the Theory and Application of Cryptology and Information Security, Kuching, Malaysia, December 2-6, 2007, Proceedings*, volume 4833 of *LNCS*. Springer, 2007.
22. Xuejia Lai and Kefei Chen, editors. *Advances in Cryptology - ASIACRYPT 2006, 12th International Conference on the Theory and Application of Cryptology and Information Security, Shanghai, China, December 3-7, 2006, Proceedings*, volume 4284 of *LNCS*. Springer, 2006.
23. Stefan Lucks. Design principles for iterated hash functions. Cryptology ePrint Archive, Report 2004/253, 2004.
24. Stefan Lucks. A failure-friendly design principle for hash functions. In Bimal K. Roy, editor, *ASIACRYPT*, volume 3788 of *LNCS*, pages 474–494. Springer, 2005.
25. Krystian Matusiewicz, Thomas Peyrin, Olivier Billet, Scott Contini, and Josef Pieprzyk. Cryptanalysis of FORK-256. In Biryukov [9], pages 19–38. See also `http://www.ics.mq.edu.au/~kmatus/FORK/`.
26. Florian Mendel, Christian Rechberger, and Martin Schläffer. Collisions for round-reduced LAKE. Submitted, 2008.
27. Florian Mendel and Vincent Rijmen. Cryptanalysis of the Tiger hash function. In Kurosawa [21], pages 536–550.
28. Moni Naor and Omer Reingold. From unpredictability to indistinguishability: A simple construction of pseudo-random functions from MACs (extended abstract). In Krawczyk [20], pages 267–282.
29. NIST. FIPS 180-2 secure hash standard, 2002.
30. NIST. Cryptographic hash project, 2007. `http://www.nist.gov/hash-competition`.
31. Sean O'Neil. Algebraic structure defectoscopy. `http://defectoscopy.com/`.
32. Sean O'Neil. Algebraic structure defectoscopy. In *Special ECRYPT Workshop – Tools for Cryptanalysis*, 2007. Available at `http://www.impan.gov.pl/BC/Program/conferences/07Crypt-prg.html`.
33. Pinakpani Pal and Palash Sarkar. PARSHA-256 - a new parallelizable hash function and a multithreaded implementation. In Thomas Johansson, editor, *FSE*, volume 2887 of *LNCS*, pages 347–361. Springer, 2003.
34. Thomas Peyrin. Cryptanalysis of Grindahl. In Kurosawa [21], pages 551–567.
35. Norbert Pramstaller, Christian Rechberger, and Vincent Rijmen. Breaking a new hash function design strategy called SMASH. In Bart Preneel and Stafford E. Tavares, editors, *SAC*, volume 3897 of *LNCS*, pages 233–244. Springer, 2005.
36. Jean-Jacques Quisquater and Jean-Paul Delescaille. How easy is collision search? Application to DES (extended summary). In Jean-Jacques Quisquater and Joos Vandewalle, editors, *EUROCRYPT*, volume 434 of *LNCS*, pages 429–434. Springer, 1989.

37. Bruce Schneier, John Kelsey, Doug Whiting, David Wagner, Chris Hall, and Niels Ferguson. *The Twofish Encryption Algorithm.* Wiley, 1999.
38. Robert Sedgewick, Thomas G. Szymanski, and Andrew Chi-Chih Yao. The complexity of finding cycles in periodic functions. *SIAM Journal of Computing*, 11(2):376–390, 1982.
39. David J. Wheeler and Roger M. Needham. TEA, a tiny encryption algorithm. In Bart Preneel, editor, *Fast Software Encryption*, volume 1008 of *LNCS*, pages 363–366. Springer, 1994.
40. Doug Whiting, Bruce Schneier, Stephan Lucks, and Frédéric Muller. Phelix - fast encryption and authentication in a single cryptographic primitive. Technical Report 2005/20, ECRYPT eSTREAM, 2005.

A Constants

For LAKE-256, IV corresponds to the first 64 hexadecimal digits of π, and the constants to the 65-th to the 192-th digits[5]:

$$\begin{array}{llll}
IV_0 = \texttt{243F6A88} & IV_2 = \texttt{13198A2E} & IV_4 = \texttt{A4093822} & IV_6 = \texttt{082EFA98} \\
IV_1 = \texttt{85A308D3} & IV_3 = \texttt{03707344} & IV_5 = \texttt{299F31D0} & IV_7 = \texttt{EC4E6C89}
\end{array}$$

$$\begin{array}{llll}
C_0 = \texttt{452821E6} & C_4 = \texttt{C0AC29B7} & C_8 = \texttt{9216D5D9} & C_{12} = \texttt{2FFD72DB} \\
C_1 = \texttt{38D01377} & C_5 = \texttt{C97C50DD} & C_9 = \texttt{8979FB1B} & C_{13} = \texttt{D01ADFB7} \\
C_2 = \texttt{BE5466CF} & C_6 = \texttt{3F84D5B5} & C_{10} = \texttt{D1310BA6} & C_{14} = \texttt{B8E1AFED} \\
C_3 = \texttt{34E90C6C} & C_7 = \texttt{B5470917} & C_{11} = \texttt{98DFB5AC} & C_{15} = \texttt{6A267E96}
\end{array}$$

For LAKE-512, IV corresponds to the first 128 hexadecimal digits of π ending trillion-th, and the constants to the 129-th to the 384-th digits.

$$\begin{array}{ll}
IV_0 = \texttt{57F5C7D088813AFC} & IV_4 = \texttt{F92F3FFEB7790C39} \\
IV_1 = \texttt{13908A7C25E945C0} & IV_5 = \texttt{428D3FD1A930A4EE} \\
IV_2 = \texttt{B273D634AF4635AB} & IV_6 = \texttt{A66C46E2B3255458} \\
IV_3 = \texttt{B8E6A0E2AE025B8F} & IV_7 = \texttt{F2AC54FEDE1EC2EA}
\end{array}$$

$$\begin{array}{ll}
C_0 = \texttt{0769441AD54C789F} & C_8 = \texttt{4623A40AB23A2E02} \\
C_1 = \texttt{3CB62BB721C2746E} & C_9 = \texttt{A43BA7CDFC9BCF82} \\
C_2 = \texttt{1BE973B3FF6C5EDE} & C_{10} = \texttt{D6AEBF43FB266C5E} \\
C_3 = \texttt{D9883F666CD37F6B} & C_{11} = \texttt{139363097AAB1247} \\
C_4 = \texttt{2A9572193E06AA68} & C_{12} = \texttt{2A53B4E0A95CAA01} \\
C_5 = \texttt{8AB87CA9222605F2} & C_{13} = \texttt{8D1770714B749520} \\
C_6 = \texttt{3B43E1D7013CEAC5} & C_{14} = \texttt{B3BC88DB689CA207} \\
C_7 = \texttt{DF6534E1E77E037E} & C_{15} = \texttt{C46EF39031B3E5A5}
\end{array}$$

B Test Values

For LAKE-256, compress(Null input) =

C5EB97EC 704D4816 5A1714E3 549343B4 18831B53 2FB84D85 E304A0A4 73CB9E03.

For LAKE-512, compress(Null input)=

804829AB81DA589B E9205F12A4EE3666 D23D5574793C9C32 4DB7387F53795476
653D40810DC4A3AA F14D3A5E8D14F043 9904191ADE724751 C9D033C934C9229E.

[5] Hexadecimal π digits are copied from http://www.super-computing.org/pi-hexa_current.html.