A CASE FOR A PARALLELIZABLE HASH

Alan Kaminsky and Stanisław P. Radziszowski
Department of Computer Science
Rochester Institute of Technology
Rochester, NY

ABSTRACT

On November 2, 2007, NIST (United States National Institute of Standards and Technology) announced an initiative to design a new secure hash function for this century, to be called ***SHA-3****. The competition will be open and it is planned to conclude in 2012. These developments are quite similar to the recent history of symmetric block ciphers—breaking of the DES (Data Encryption Standard) and emergence of the AES (Advanced Encryption Standard) in 2001 as the winner of a multiyear NIST competition. In this paper we make a case that* parallelizability *should be one of the properties sought in the new SHA-3 design. We present a design concept for a parallelizable hash function called* ***PHASH*** *based on a block cipher, and we discuss PHASH's performance and security.*

1. INTRODUCTION

This paper is concerned with unkeyed hash functions $H : \{0,1\}^* \rightarrow \{0,1\}^m$, where in practice the input length can be bounded by some large constant, say 2^{96}. The well known Merkle-Damgård iterated hash template is used in almost all practical hash functions. For messages spanning several blocks, the chaining mechanism forces the blocks of the message to be processed sequentially. Even at the level of the individual blocks, only small parts of each round in the compression function can be parallelized due to the sequential structure of typical compressions. This may be not a big problem for short messages, but, as we will argue, it will likely become a problem in the future when there will be an increasing need to hash very long inputs.

The two most widely used practical hash functions are MD5 [16, 12, 18] and SHA-1 [10, 12]. Both follow the sequential chaining Merkle-Damgård template, and both have a relatively large number of rounds, 64 and 80, respectively, which have to be executed in sequence. Apart from only minor possibilities of low level parallelization within each round, both designs are inherently sequential. Moreover, the new hash functions of the SHA-2 family [10] with longer hashes, which were added to the standard by NIST in 2002, and the newer block cipher based hashes Whirlpool [15, 19] and Maelstrom-0 [7], follow the same fundamental design principle, and thus all of them are equally inherently sequential.

Dramatic progress on collision finding techniques for MD5 and SHA-1 in recent years [4, 17, 20, 21] prompted NIST to announce on November 2, 2007 a new initiative to design a secure hash function for this century, to be called **SHA-3** [11]. The competition will be open and it is planned to conclude in 2012 [11]. These developments are quite similar to the recent history of symmetric block ciphers—breaking of the DES (Data Encryption Standard) and emergence of the AES (Advanced Encryption Standard) in 2001 as the winner of a multiyear NIST competition.

In this paper we argue that *parallelizability* of the new SHA-3 should be stressed strongly during the competition. Section 2 presents use cases motivating a parallelizable hash function. Section 3 presents a design concept for a parallelizable hash function called "PHASH." Section 4 compares PHASH with related work. Section 5 discusses the performance of PHASH. Section 6 discusses the security of PHASH.

2. USE CASES

The DES block cipher has been used for over 30 years since its standardization in 1977; it was the NIST approved standard block cipher for nearly 25 years until its replacement by the AES block cipher in 2001. A new standard hash function can be expected to be used for a similar span of years. Computers nowadays—including most of our students' laptops—have multicore CPUs and nearly a terabyte of disk space. A hash function that must be computed with a sequential algorithm, as all Merkle-Damgård iterated hash functions must be, cannot utilize the full performance of multiprocessor machines and is not a good choice for a hash function that will be used for many years. We argue that the

978–1–4244–2677–5/08/$25.00

SHA-3 initiative should strongly encourage the development of parallelizable hash functions.

For short messages (say up to 1 Mb) conventional hash functions seem fast enough, even on sequential machines. Parallelizability is much more critical for large inputs; it will make a great difference when hashing many terabytes of data. For example, we can foresee use cases like these:

- A consumer hashes a two-hour high resolution movie to see if it was downloaded correctly and to verify the digital signature on the hash of the movie (as part of a digital rights management system, perhaps). The same applies to all kinds of content, such as games and multimedia.

- A laboratory hashes a massive protein sequence database and adds a digital signature. A scientist downloads the database, hashes it, verifies the integrity of the download, and verifies the signature to authenticate the database's origin. The same applies to any large collection of data, such as global financial records databases and large scale computational science simulations.

- A user hashes the entire file system while backing it up, so that five years later the user can instantly check whether the backup was corrupted.

- A computer's hard disk is seized by police for a criminal investigation. An image of the disk is created and the hash of the entire disk image is computed. As copies of the disk image are made, hashes of the copies are used to prove chain of custody. Forensics analysts can testify in court that the disk image copies they analyzed were the same as the original, because their hashes were the same. [13]

Parallelizability is also critical when hash functions have to be computed at very high speeds—for example, to compute hashed message authentication codes (HMACs) in hardware on very high speed networks. The hash calculations may not be able to keep up with the data stream unless multiple blocks of the data stream can be hashed in parallel by multiple processors.

These examples use a hash function for two separate goals, *checksumming* (to verify the data's integrity) and *authentication*, as part of an HMAC or digital signature (to verify the data's origin). Mathematically, both operations are the same, and the only difference is the size of the data. Checksumming is usually associated with large data sets, authentication with short messages.

Some maintain that since functions for checksumming already exist (such as CRCs), hash functions need not be used for checksumming large inputs. Thus, hash functions can be confined to authenticating short inputs, and so there is no need for SHA-3 to be a parallelizable hash function that is efficient on large inputs.

We argue that in practice, hash functions are in fact used for checksumming, and that this is not likely to change in the future. Recognizing this, hash functions in the current Secure Hash Standard are already designed to handle large input sizes—2^{64} bits for SHA-1 and SHA-256, 2^{128} bits for SHA-384 and SHA-512. However, actually trying to compute the SHA-1 hash of a two-million-terabyte (2^{64}-bit) file in the sequential chaining mode would be suicide. A parallelizable SHA-3 is crucial for hashing even the already-standardized input sizes. In addition, application designers have an easier job if they don't have to use different functions for checksumming and for authentication. Just compute SHA-3(x) for any x and no more needs to be done.

Others argue that, while hashing large messages is important, the hash function design itself should concentrate on *compression* only. Hashing large messages should be addressed by designing one or more *modes of operation* for the hash function (analogous to the modes of operation for a block cipher, such as CBC mode or counter mode). These two components, compression and mode of operation, should be designed separately, and so, again, there is no need to consider parallelizability when designing the hash function for SHA-3.

We consider this to be a risky and shortsighted approach. Whenever a hash function deals with more than one message block it must define how to combine the results of compression on multiple blocks. Almost all hash applications require more than one block. We cannot leave the mode of operation unspecified for SHA-3, as was done for block ciphers, because the results of compressing each block must somehow be merged together to yield a single hash value. Hence, once SHA-3 defines how to merge multiple blocks it will be too late to reconsider the mode without ending up with an awkward design with different modes for different data. Both the compression function and the mode of operation must be defined in the SHA-3 standard, and the total design must be parallelizable to be able to hash large messages efficiently.

3. PARALLEL HASH FUNCTION

Here we propose a design concept for a hash function that can be computed in parallel, dubbed "PHASH." The PHASH computation comprises three phases: message padding and length encoding, compression, and reduction. The compression phase uses a compression function as a

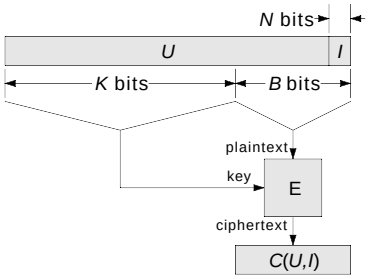

Figure 1. The PHASH compression function

building block. We describe the compression function first, then the three phases of the PHASH computation.

Compression function. Following Biham [3], PHASH uses a block cipher for message compression; the size of the hash value is the same as the cipher's block size. Any secure block cipher can be used. The cipher takes a K-bit key and a B-bit plaintext block as input and produces a B-bit ciphertext block as output. This will be denoted as $ciphertext = E(key, plaintext)$, where E stands for "encrypt."

The PHASH compression function (Figure 1) takes two inputs: a $(K+B-N)$-bit uncompressed value U and an N-bit counter I. The compression function produces a B-bit compressed value $C(U,I)$. Let $U[a..b]$ denote bits a through b inclusive of U, where bits are numbered from most to least significant starting at 0. The compression formula is

$$C(U,I) = E(U[0..K-1], U[K..K+B-N-1] || \langle I \rangle) \qquad (1)$$

That is, the first K bits of U are the block cipher key; the last $(B-N)$ bits of U are concatenated with $\langle I \rangle$, the N-bit encoding of the counter, to form the B-bit plaintext; and the block cipher's output ciphertext yields the B-bit compressed value.

PHASH can use either of two recent block ciphers specifically proposed for hash functions: W, the block cipher of Whirlpool [15], and $\mathscr{M}$, the block cipher of Maelstrom-0 [7]. W uses K = 512 and B = 512. $\mathscr{M}$ uses K = 1024 and B = 512.

Another possibility is to use the Rijndael block cipher [5]. When originally proposed as an AES candidate, Rijndael supported block sizes of 128, 192, and 256 bits; however, only the 128-bit block size was standardized as AES. We refer to Rijndael with a 256-bit block size as "AESplus." AESplus uses K = 256 and B = 256.

NIST is seeking hash sizes of 224, 256, 384, and 512 bits for SHA-3. PHASH with W or $\mathscr{M}$ yields a 512-bit hash; that value can be truncated for a 384-bit hash. PHASH with AESplus yields a 256-bit hash; that value can be truncated for a 224-bit hash.

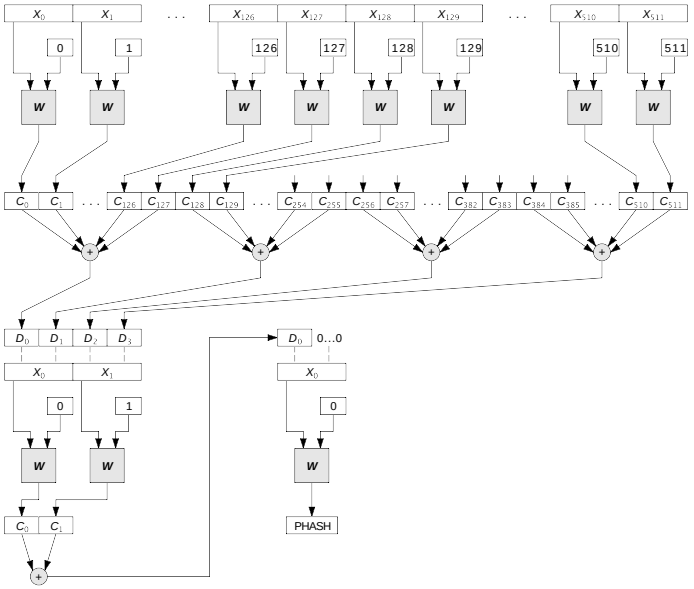

Figure 2. PHASH computation tree with cipher W, $L = 512$, and $R = 128$

Message padding and length encoding phase. Like SHA-1 and the SHA-2 family, the PHASH computation begins by padding the message to be hashed and adding the message length.

Let N, the number of bits in the counter, also be the number of bits used to encode the message length. SHA-1 and SHA-256 use N = 64; SHA-384 and SHA-512 use N = 128. For these hash functions the choice of N does not affect the actual hash computation. However, as will be seen, for PHASH it does.

Let M, a bit string, be the message to be hashed. Let $|M|$ be the length of M in bits. $|M|$ must be less than 2^N. Append the following bits to M: a 1 bit; zero or more 0 bits; and the value of $|M|$ expressed as an N-bit binary number, most significant bit first; such that the resulting bit string's length is a multiple of $(K+B-N)$. Call the resulting bit string X.

Compression phase. Split X into $(K+B-N)$-bit blocks. Let there be L such blocks; $L = |X|/(K+B-N)$. Let X_i be the i-th block. For each block X_i, compute the compressed block C_i:

$$C_i = C(X_i, i) \qquad (2)$$

That is, each message block is compressed together with its own block number as the counter. At the end of the message compression phase we have L compressed blocks $C_i, 0 \le i \le L-1$.

Note that if the number of bits used to encode the message length (N) is increased, thus increasing the size of the counter, each application of C will absorb fewer bits of the

message, thus increasing the hash computation time. N must be chosen to give the desired tradeoff between the hash's security, computation time, and the maximum allowed message length.

Reduction phase. Following Bellare *et al.* [1, 2], PHASH compresses all the message blocks X_i independently, then combines the compressed blocks C_i together. This provides parallelizability and incrementality. However, PHASH uses a different reduction scheme designed to improve the hash's security.

The PHASH reduction phase is as follows. If after the compression phase there is only one compressed block C_0, then C_0 is the hash of the original message. Otherwise, compute reduced blocks D_j from the compressed blocks C_i:

$$D_j = \bigoplus_{i=R\cdot j}^{R\cdot j+R-1} C_i, \quad 0 \le j \le \lceil L/R \rceil - 1 \tag{3}$$

taking $C_i = 0$ for $i \ge L$. That is, for each group of R consecutive compressed blocks, the blocks in the group are bitwise XORed together. This yields $\lceil L/R \rceil$ reduced blocks D_j. Concatenate these reduced blocks, and append zero or more 0 bits such that the resulting bit string's length is a multiple of $(K+B-N)$. Replace the original bit string X with this new bit string. Apply the compression and reduction phases to this new X. Keep repeating the compression and reduction process until the result is a single compressed block, and that is the hash of the original message.

The PHASH computation can be depicted as a tree (Figure 2). At each level of the tree, each group of R input blocks X_i is reduced into a single output block D_j. For security, the final reduced block D_0 is run through the compression function (encrypted) one last time, unless the final reduced block consists of just one compressed block C_0.

Example. Consider an example of PHASH using $\mathscr{M}$, with K = 1024 bits, B = 512 bits, counter size N = 96 bits, and R = 128. At each level of the reduction tree, an input string X of 184,320 bits (= $128\times(1024+512-96)$ bits) yields an output string D of 512 bits, a reduction factor of 360. A message of up to 184,223 bits (about 22 kilobytes) requires only one level in the tree (not counting the final compression). A message of up to 66,355,103 bits (about 8 megabytes) requires two levels in the tree. A maximum-size message of $2^{96}-1$ bits requires 11 levels in the tree.

Implementation. While PHASH is designed to be parallelizable, it can easily be implemented as a sequential program. PHASH can even be implemented in streaming mode, without requiring the entire message to be stored in memory.

The PHASH algorithm can be naturally realized in hardware, in conventional programming languages such as C and assembly, and in object oriented programming languages such as C++ and Java. Since modern block ciphers (including AES, W, and $\mathscr{M}$) are designed to be implemented on low-end 8-bit processors as well as high-end 32- and 64-bit processors, PHASH can be implemented on low-end and high-end processors as well.

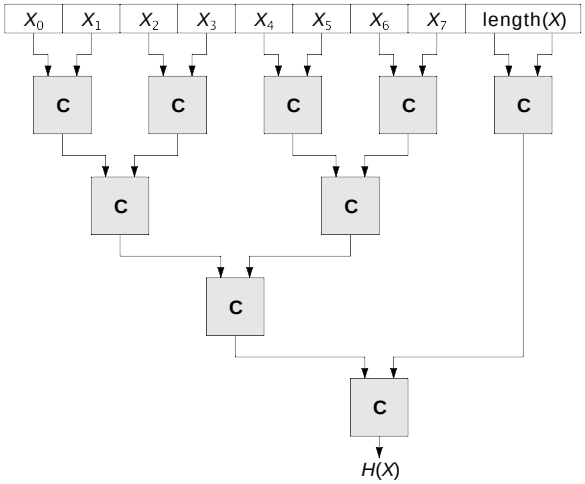

Figure 3. Damgård's parallel hash

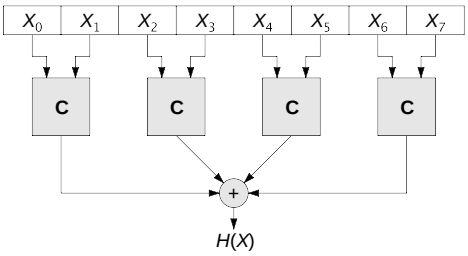

Figure 4. XHASH

4. RELATED WORK

Parallelizable hashes. Damgård's 1989 paper [6] briefly describes a design for a parallelizable hash function. Two more recent papers discuss parallelizable hash function designs in detail: one by Bellare, Guérin and Rogaway [1] in 1995 and one by Bellare and Micciancio [2] in 1997. The latter paper, focusing on unkeyed parallelizable hashes, is more directly relevant to our work; the former paper focuses on keyed MACs.

The parallelizable hash function in [6] is based on a compression function that maps a $2B$-bit input into a B-bit output. The hash function is simply a binary tree of compression functions (Figure 3). At each level of the tree, the compression functions can all be computed in parallel. The output of the tree is compressed together with the hash of the message length, yielding the hash of the input message.

In [2], the authors study a very simple and tempting idea called XHASH, where the hashes of individual blocks are combined with XOR (Figure 4). This is clearly parallelizable. Unfortunately, it is shown that if the number of blocks is at least as large as the number of bits in the hash, one can, with high probability, construct a preimage attack on XHASH. Other ways of combining the hashes of individual blocks are also discussed in [2], but apparently none of them was studied further in the literature, perhaps because their security relies on far from well understood difficulty of finding short vectors in integer lattices, certain weighted knapsack problems, or special group theoretical problems.

PHASH uses the main parallelizability idea from XHASH, with reduction via XOR. To guarantee security it is sufficient to limit the number of blocks combined in a single XOR; this foils the XHASH attack described in [2]. We prevent this attack by re-encrypting the intermediate results with the block cipher every R blocks. $R = 128$ seems a reasonable choice but could be reduced to increase security. In contrast to Damgård's parallel hash, PHASH has a much shallower reduction tree, the number of levels being about the base-360 logarithm of the number of input blocks (for the example given earlier) as opposed to the base-2 logarithm. Since the levels of the reduction tree have to be computed sequentially, PHASH requires less computation time than Damgård's parallel hash if both use a similar compression function.

Compression: block cipher vs. special purpose function. The compression functions in essentially all practical hashes were designed especially for those hash functions. Almost every cryptography textbook will briefly discuss the possibility of obtaining hash compression from a block cipher, but such suggestions are typically quickly dismissed for two reasons [18]: the strings produced by block ciphers are too short, and block ciphers are too slow. Other researchers argue in addition [8] that special-purpose hash compression functions permit more flexibility in the design, since we don't need to worry about secret keys and invertibility under each fixed key.

We respond to these criticisms of block-cipher-based hash functions in general, and PHASH in particular, as follows. Since PHASH can use a block cipher with a 512-bit block such as W or $\mathscr{M}$, PHASH meets the hash size requirement of SHA-3. Speed is not an issue; the block-cipher-based Whirlpool hash function has been shown to be faster than SHA-512 on a sequential machine [9, 14]. Because PHASH absorbs more input bits than Whirlpool on each application of the block cipher, PHASH on a sequential machine will be faster still, and will be orders of magnitude faster on a parallel machine.

As for the potential of more flexibility in the design of hashes using special-purpose compression functions, so far this approach has proved disastrous. We see successful attacks on MD5 and SHA-1 because of poor designs of the compression functions. Even more, Biham [3] has argued that we should use the much better understood block cipher design principles in hashes. Doing so would easily prevent all current attacks, and there is no evidence that cipher-based compression should be slower. As it is, it seems that current hash compression functions try to hide their weaknesses by employing a large number of rounds, instead of using fewer but more powerful rounds as block ciphers do.

5. PERFORMANCE OF PHASH

Following some general observations about PHASH's performance, some initial throughput measurements for PHASH are reported below. A thorough study of optimized hardware and software implementations still needs to be done for PHASH, as was done for Whirlpool [9, 14].

Sequential performance. The performance of PHASH using W can be determined relative to the performance of Whirlpool. While Whirlpool absorbs only 512 bits of the message with each application of W, PHASH absorbs 928 bits. For large inputs, the time needed for the reduction phase is negligible compared to the compression phase. Thus, PHASH using W should run nearly twice as fast as Whirlpool on a sequential machine.

Parallel performance. If multiple processors are available, and if the original message blocks X_i can be accessed directly (for example, if the message is stored in a file), the compressed blocks C_i can be computed in parallel. Since each block is compressed independently, this is a "massively parallel" computation that requires no synchronization or communication between the processors, resulting in a speedup almost equal to the number of processors. Likewise, at each level of the reduction tree the reduced blocks D_j can be computed in parallel. The reduction tree levels themselves must be computed in sequence. However, since the number of levels grows only as the base-360 logarithm of the message size, the reduction computation occupies a decreasing fraction of the total computation as the message size increases. A parallel implementation of PHASH should therefore yield very good performance as the message sizes scale up.

Incremental re-hashing. Besides parallelizability, another nice property of PHASH (not seen in standard hashes) is that small length preserving changes in the message permit a very fast update of the hash value by recomputing only the affected intermediate quantities along paths of a very

shallow reduction tree. This would require caching the D_j blocks at each level of the reduction tree; however, the length of these blocks is much smaller than the length of the original message.

Throughput measurements. One hardware implementation and performance study of PHASH has been done [22]. Sequential versions of three hash functions—SHA-512, Whirlpool, and PHASH—were implemented on a Xilinx Vertex-5 LX330 FPGA. All three hash functions generated a 512-bit hash. The PHASH implementation used the W block cipher (which Whirlpool also uses) with a counter size N of 128 bits. The SHA-512 implementation attained a throughput of 1.8 Gbps; Whirlpool, 7.7 Gbps; and PHASH, 15.1 Gbps. Sequential PHASH did in fact run nearly twice as fast as Whirlpool and ran over eight times faster than SHA-512.

Parallel versions of PHASH with 2, 4, 8, and 16 instances of W were also implemented. These attained throughputs of 28.9, 55.6, 101.0, and 182.6 Gbps, respectively. The speedups relative to sequential PHASH were 1.9, 3.7, 6.7, and 12.1, respectively.

6. SECURITY OF PHASH

While a formal proof of PHASH's security still needs to be done, some simple observations can be made.

Block cipher security. First, our design is quite close to that considered by Bellare and Micciancio [2]. Roughly speaking, if we trust the security of the block cipher used in PHASH's compression function, then we should trust the security of PHASH, provided the block cipher is not used in such a way that PHASH can be attacked without breaking the block cipher.

XHASH attack. The attack against XHASH in [2] is likely to succeed if R, the number of message blocks XORed together, is close to B, the number of bits in the hash value. The attack relies on finding a linear dependency among R random vectors in $GF(2^B)$. For PHASH, the probability of finding a linear dependency among 128 random vectors in $GF(2^{512})$ is vanishingly close to zero, so PHASH should not be susceptible to the XHASH attack. Note that PHASH proffers a tradeoff in the choice of R: increasing R reduces the computation time, since fewer encryptions are needed in the reduction phase, but also increases the likelihood of a successful XHASH attack. A choice of $R = 256$ may still be secure enough; a choice of $R = 64$ or 32, while more secure, would increase the computation time somewhat.

Preimage resistance. Like all hash functions based on block ciphers, finding the input to PHASH that yielded a given output reduces to finding the plaintext and key that yield a known ciphertext. If PHASH's block cipher is resistant to a ciphertext-only attack, it seems that PHASH is preimage resistant.

Second preimage and collision resistance. It is possible to find second preimages and collisions in PHASH in 2^N operations, where N is the number of bits in the counter. Consider PHASH computing the hash of a message with two input blocks X_0 and X_1. The high-order bits of X_0 give a key for the block cipher; the low-order bits of X_0 concatenated with a counter of 0 give a plaintext; the resulting ciphertext is the compressed block C_0. Similarly, X_1 yields C_1. The reduced block D_0 is $C_0 \oplus C_1$. Now, pick a different input block X_1'; compress that to yield C_1'; and compute $C_0' = C_1' \oplus D_0$. If we can find an input block X_0' that compresses to C_0', then the message $(X_0'||X_1')$ will be a second preimage having the same hash value as the original message $(X_0||X_1)$. To find X_0', pick an arbitrary key, decrypt C_0', and see if the low-order N bits of the resulting plaintext have the required counter value (0); if so, X_0' is the key concatenated with the high-order bits of the plaintext. If the block cipher behaves as a random mapping, the probability of success is 2^{-N}. Collisions can be found in a similar fashion.

Thus, the counter size N can be used to tune PHASH's security level. A larger counter size gives an increased security level, at the price of increased hash computation time. For SHA-3, a B-bit hash function is required to have a security level of $B/2$ bits against collision attacks [11]; this would require PHASH's counter to be $B/2$ bits long. Also for SHA-3, a B-bit hash function is required to have a security level of $B-m$ bits against second preimage attacks when the input message lengths are 2^m bits or less [11]; this would require PHASH's counter to be $B-m$ bits long.

PHASH can achieve all these required security levels simply by making the counter size N equal to the cipher's block size B. Parameterized this way, PHASH operates similarly to a block cipher in counter mode; successive compressed blocks C_i are the encryptions of successive counter values, with the encryption keys being determined by the input message bits. We feel, however, that this buys only a small amount of additional security at an excessive price in performance.

Length extension attack. PHASH is not susceptible to a length extension attack. Since PHASH's final value is always the result of an encryption, it is not possible to construct the hash of a message M_2 given the hash of a message M_1 but not M_1 itself, where M_2 consists of M_1 (including padding and message length) followed by additional data. To carry out a length extension attack would require breaking the block cipher.

7. CONCLUSION

Parallelizable hash functions will be needed to hash the increasingly-large inputs we foresee in the coming years. While PHASH's security has not yet been rigorously analyzed, the PHASH design concept shows that it is possible to design an efficient, secure, *parallelizable* hash function. Any future standard hash function should allow parallel implementations.

In the SHA-3 competition announcement, "flexibility" is listed as the last criterion that will be used to judge the candidate hash functions, and the following is stated as one example of flexibility: "Implementations of the algorithm can be parallelized to achieve higher performance efficiency" [11]. We feel that parallelizability should be stressed much more strongly in the competition. A hash algorithm with an inherent limit on its performance—one that cannot experience a speedup on a parallel computer—is inferior to a hash algorithm that does not have this limit, assuming the hashes' security levels are equal. SHA-3 proposals that allow parallel implementations should be judged superior to non-parallelizable hash functions.

REFERENCES

[1] M. Bellare, R. Guérin, and P. Rogaway. XOR MACs: new methods for message authentication using finite pseudorandom functions. In *Advances in Cryptology—CRYPTO '95*, 15–28.

[2] M. Bellare and D. Micciancio. A new paradigm for collision-free hashing: incrementality at reduced cost. In *Advances in Cryptology—EUROCRYPT '97*, 163–192.

[3] E. Biham. Recent advances in hash functions: the way to go. June 24, 2005. `http://www.cs.technion.ac.il/~biham/Reports/Slides/hash-func-krakow-2005.ps.gz`

[4] J. Black, M. Cochran, and T. Highland. A study of the MD5 attacks: insights and improvements. In *Proceedings of the 13th International Workshop on Fast Software Encryption*, 2006, 262–277.

[5] J. Daemen and V. Rijmen. AES submission document on Rijndael, Version 2, September 1999. `http://csrc.nist.gov/CryptoToolkit/aes/rijndael/Rijndael.pdf`

[6] I. Damgård. A design principle for hash functions. In *Advances in Cryptology—CRYPTO '89*, 416–427.

[7] D. Gazzoni Filho, P. Barreto, and V. Rijmen. The Maelstrom-0 hash function. In *Proceedings of the 6th Brazilian Symposium on Information and Computer Systems Security*, 2006.

[8] V. Klima, posts at the NIST Cryptographic Hash Forum, 2007. `hash-forum@nist.gov`

[9] M. McLoone, C. McIvor, and A. Savage. High-speed hardware architectures of the Whirlpool hash function. In *Proceedings of the 2005 IEEE International Conference on Field-Programmable Technology*, 147–153.

[10] National Institute of Standards and Technology. Secure hash standard. *FIPS 180-2*, 2002. `http://csrc.nist.gov/publications/fips/fips180-2/fips180-2.pdf`

[11] National Institute of Standards and Technology. Announcing request for candidate algorithm nominations for a new cryptographic hash algorithm (SHA-3) family. *Federal Register*, 72(212):62212–62220, November 2, 2007. `http://csrc.nist.gov/groups/ST/hash/federal_register.html`

[12] A. Menezes, P. van Oorschot, and S. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.

[13] J. Patzakis. Maintaining the digital chain of custody. In *Infosecurity Europe Conference*, 2003. `http://www.infosec.co.uk/files/guidance_software_04_12_03.pdf`

[14] N. Pramstaller, C. Rechberger, and V. Rijmen. A compact FPGA implementation of the hash function Whirlpool. In *Proceedings of the 2006 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, 2006, pages 159–166.

[15] V. Rijmen and P. Barreto. The Whirlpool Hash Function, 2003. `http://paginas.terra.com.br/informatica/paulobarreto/WhirlpoolPage.html`

[16] R. Rivest. The MD5 message-digest algorithm. Internet RFC 1321, 1992. `http://www.ietf.org/rfc/rfc1321.txt`

[17] A. Satoh. Hardware architecture and cost estimates for breaking SHA-1. In *Proceedings of the 8th International Conference on Information Security (ISC 2005)*, 2005, pages 259–273.

[18] B. Schneier. *Applied Cryptography, Second Edition.* John Wiley & Sons, 1996.

[19] W. Stallings. The Whirlpool secure hash function. *Cryptologia*, 30(1):55–67, 2006.

[20] X. Wang and H. Yu. How to break MD5 and other hash functions. In *Advances in Cryptology—EUROCRYPT 2005*, 19–35.

[21] X. Wang, Y. Yin, and H. Yu. Finding collisions in the full SHA-1. In *Advances in Cryptology—CRYPTO 2005*, 17–36.

[22] P. Zalewski. FPGA design and performance analysis of SHA-512, Whirlpool and PHASH hashing functions. Rochester Institute of Technology Computer Engineering M.S. Thesis, May 2008.