Square Hash: Fast Message Authentication via Optimized Universal Hash Functions*

Mark Etzel[1], Sarvar Patel[1], and Zulfikar Ramzan**[2]

[1] Bell Labs, Lucent Technologies
{mhetzel,sarvar}@bell-labs.com
[2] Laboratory for Computer Science, MIT
zulfikar@theory.lcs.mit.edu

Abstract. This paper introduces two new ideas in the construction of fast universal hash functions geared towards the task of message authentication. First, we describe a simple but novel family of universal hash functions that is more efficient than many standard constructions. We compare our hash functions to the MMH family studied by Halevi and Krawczyk [12]. All the main techniques used to optimize MMH work on our hash functions as well. Second, we introduce additional techniques for speeding up our constructions; these techniques apply to MMH and may apply to other hash functions. The techniques involve ignoring certain parts of the computation, while still retaining the necessary statistical properties for secure message authentication. Finally, we give implementation results on an ARM processor. Our constructions are general and can be used in any setting where universal hash functions are needed; therefore they may be of independent interest.

Key words: Message authentication codes, Universal Hashing.

1 Introduction

Message Authentication. Designing good Message Authentication schemes is a very important objective in cryptography. The goal in message authentication is for one party to efficiently transmit a message to another party in such a way that the receiving party can determine whether or not the message he receives has been tampered with. The setting involves two parties, Alice and Bob, who have agreed on a pre-specified secret key x. Two algorithms are used: an algorithm S_x that applies a tag to a message, and a verification algorithm V_x that checks if the tag associated with a given message is valid. If Alice wants to send a message M to Bob, she first computes a *message authentication code*, or MAC, $\mu = S_x(M)$. She sends (M, μ) to Bob, and upon receiving the pair, Bob computes $V_x(M, \mu)$ which returns 1 if the MAC is valid, or returns 0 otherwise. Without knowledge of the secret key x, it should be infeasible for an adversary

* Extended abstract. A full version is available at
http://theory.lcs.mit.edu/~zulfikar

** Work done while this author was at Lucent Technologies. This author would like to acknowledge DARPA grant DABT63-96-C-0018 and an NSF graduate fellowship.

to construct a message and a corresponding MAC that the verification algorithm will accept as valid. The formal security requirement for a MAC was first defined by Bellare, et al [4]. This definition is analogous to the formal security definition of a digital signature [11]. In particular, we say that an adversary forges a MAC if, when given oracle access to (S_x, V_x), where x is kept secret, the adversary can come up with a pair (M^*, μ^*) such that $V_x(M^*, \mu^*) = 1$ but the message M^* was never made an input to the oracle for S_x.

IMPORTANCE OF EFFICIENT MACS. In general, MACs are computed frequently and on inputs which are often thousands of bytes long. Moreover, computing and verifying tags is typically done in software, and may be done on relatively weak platforms. Additionally, the computations must be done in real time. Therefore, developing techniques for optimizing MAC Algorithms while retaining the appropriate level of security is crucial. This paper presents two novel ideas in this direction.

COMMON APPROACHES TO MESSAGE AUTHENTICATION. One approach to message authentication involves using a secure block cipher, such as DES [21], in cipher block chaining mode. Another approach to message authentication, often seen in practice, involves using cryptographic hash functions like $MD5$ [24]. For example, one approach was to set $\mu = MD5(x \cdot m \cdot x)$; unfortunately, this particular scheme is vulnerable to a clever key recovery attack due to Preneel and and van Oorschot [23]. Other work on using cryptographic hash functions in MACs is the HMAC construction of Bellare, et al [3]; their schemes are good because they use fast and secure cryptographic building blocks. At first it appears that these techniques yield the most efficient results; however, Wegman and Carter [28] discovered that *universal hash functions*, allow us to avoid using heavy duty cryptographic primitives on the entire input string.

THE UNIVERSAL HASH FUNCTION APPROACH. In this approach, one starts with a family of hash functions H where for any pair $m \neq m'$ and for any element δ in the range, $\Pr_h[h(m) - h(m') = \delta] \leq \epsilon$. Here ϵ is a parameter related to the security of the MAC). Such functions are called ϵ-Δ-universal hash functions. Now, in order to compute the authentication tag for a message m, the communicating parties secretly agree on a function $h \in H$ chosen at random, and on a sequence of random pads $p_1, p_2 \ldots$. To compute a MAC on the i-th message m_i, the sender computes $\mu_i = h(m_i) + p_i$. One remarkable aspect of this approach is that, even if a *computationally unbounded* adversary performs q black-box oracle queries to both algorithms used by the MAC, he has probability less than $q\epsilon$ to forge the MAC. The idea in the Wegman-Carter construction is to pre-process a message quickly using universal hash functions, and then apply a cryptographic operation such as a one-time pad. In general, the one-time pad can be replaced by pseudo-random sequence. Then, the parties would have to pre-agree on the function h and on the seed s which would be fed to either a pseudo-random generator or a pseudo-random function [10]. This approach to message authentication was first studied in [6]. If pseudo-randomness is used, then the resulting MAC is secure against a polynomially bounded adversary.

The Square Hash. This paper introduces two new ideas in the construction of fast universal hash functions. We start with a simple but novel family of universal hash functions which should be more efficient than certain well-known hash function families. The efficiency lies in the fact that whereas other common constructions involve integer multiplications, our construction involves squaring integers. Since squaring a large integer requires fewer basic word multiplications than multiplying two large integers, we get a speed-up. In certain architectures, multiplication takes significantly more time than other basic arithmetic operations, so we can get good savings with this approach. Our second idea is to optimize the implementation of this hash function by ignoring certain parts of the computation; moreover, we formally prove that, despite ignoring these parts of the computation, the bound ϵ on the resulting optimized hash function is still low enough to provide for a very secure MAC. One can think of this as "theoretical hacking." Specifically, the second new idea in this paper is to completely ignore some of the carry bits when performing the computation of the hash function in our basic construction. Since carry bits can be cumbersome to deal with, we can save computational effort in this manner. We stress that this savings will primarily occur when our tag size is several words long since some architectures allow you to multiply two words, and get a two-word result with all the carries "for free." At first it seems counterintuitive that we can simply ignore what appears to be a crucial part of the computation. However, we are able to obtain a bound on the resulting value of ϵ and we show that our MAC algorithms are still secure for natural choices of the parameters.

Square Hash builds on some of the ideas in the MMH construction of Halevi and Krawczyk [12]; Knudsen independently proposed a similar construction for use in block cipher design [15]. We start with an underlying hash function which is similar to the one used in MMH; however, our new hash function performs fewer multiplications. In MMH, the final carry bit of the output is ignored – in Square Hash we extend this idea by showing that we can ignore almost all of the carry bits and can still get quite a reasonable trade-off in security. Hence, in theory, Square Hash should be a strong alternative to MMH. We have implementation results on an ARM processor to substantiate this claim. Moreover, since word blocks in the input can be worked on independently, our constructions are parallelizable. We also show how to efficiently convert any Δ-universal hash function into a strongly universal hash function. Thus Square Hash has other applications besides those related to message authentication.

Previous Work. Unconditionally secure message authentication was first studied in [9] and later in [28]. The universal hash function approach for MACs was first studied in [28] and the topic has been heavily addressed in the literature [27], [16], [25], [2], [13], [14], [26], [12]. The MMH scheme [12] is our point of departure. MMH achieves impressive software speeds and is substantially faster than many current software implementations of message authentication techniques and software implementations of universal hashing. Unfortunately, it is impossible to do precise comparisons because the available data represents simulations done on various platforms. The reader can refer to [26], [16], [5], [20] for implementation results of various MAC schemes. This paper aims to extend the ideas in the MMH construction by exhibiting, what seems to be, a faster to compute underlying hash function and by developing some new ideas which noticeably optimize the implementation with reasonably small security costs. In addition to their message authentication applications, universal hash functions are used in numerous other settings [19], [22], [7], [28].

Organization of This Paper. In section 2 we review the basic definitions and properties of universal hash function families and their variants. In section 3 we give the basic construction of the Square Hash, prove some basic properties, and explain why it should perform better than one of the most well-known universal hash function families. In section 4 we compare Square Hash with the MMH family of hash functions [12]. We show that all of their clever optimization techniques apply to Square Hash. We examine a novel optimization technique of ignoring all the carry bits in certain parts of the computation, and prove that the resulting construction still yields a strong message authentication scheme. Finally, in the last two sections we discuss some relevant implementation issues and give implementation results on an ARM processor.

2 Preliminaries

Let H be a family of functions going from a domain D to a range R. Let ϵ be a constant such that $1/|R| \leq \epsilon \leq 1$. The probabilities denoted below are all taken over the choice of $h \in H$.

Definition 1. *H is a universal family of hash functions if for all $x, y \in D$ with $x \neq y$, $\Pr_{h \in H}[h(x) = h(y)] = 1/|R|$. H is an ϵ-almost-universal family of hash functions if $\Pr_{h \in H}[h(x) = h(y)] \leq \epsilon$.*

Definition 2. *Let R be an Abelian Group and let $'-'$ denote the subtraction operation with respect to this group. Then H is a Δ-universal-family of hash functions if for all $x, y \in D$ with $x \neq y$, and all $a \in R$, $\Pr_{h \in H}[h(x) - h(y) = a] \leq 1/|R|$. H is ϵ-almost-Δ-universal if $\Pr_{h \in H}[h(x) - h(y) = a] \leq \epsilon$.*

Definition 3. *H is a strongly universal family of hash functions if for all $x, y \in D$ with $x \neq y$ and all $a, b \in R$, $\Pr_{h \in H}[h(x) = a, h(y) = b] \leq 1/|R|^2$. H is ϵ-almost-strongly-universal family of hash functions if $\Pr_{h \in H}[h(x) = a, h(y) = b] \leq \epsilon/|R|$.*

3 Square Hash

We now describe some basic constructions of universal hash function families based on squaring. We also examine modifications that enable faster implementations at negligible costs in collision probability. In the definitions and theorems that follow, we work over the integers modulo p where p is a prime.

3.1 The Basic Construction

Definition 4. *Define the SQH family of functions from Z_p to Z_p as: $SQH \equiv \{h_x : Z_p \longrightarrow Z_p | x \in Z_p\}$ where the functions h_x are defined as:*

$$h_x(m) \equiv (m + x)^2 \bmod p. \tag{1}$$

Theorem 1. *The family SQH is Δ-universal.*

Proof. For all $m \neq n \in Z_p$, and $\delta \in Z_p$: $\Pr_x[h_x(m) - h_x(n) = \delta] = \Pr_x[(m + x)^2 - (n + x)^2 = \delta] = \Pr_x[m^2 - n^2 + 2(m - n)x = \delta] = 1/p$. The last inequality follows since for all $m \neq n \in Z_p$ and $\delta \in Z_p$ there is a unique x which satisfies the equation $m^2 - n^2 + 2(m - n)x = \delta$. □

3.2 Converting From Delta-Universal to Strongly Universal

We now show how to convert any Δ-universal family of hash functions to a strongly universal family of hash functions.

Definition 5. *Define the SQHU family of functions from Z_p to Z_p as: SQHU $\equiv \{h_{x,b} : Z_p \longrightarrow Z_p | x, b \in Z_p\}$ where the functions $h_{x,b}$ are defined as:*

$$h_{x,b}(m) \equiv ((m + x)^2 + b) \bmod p. \tag{2}$$

Theorem 2. *The family SQHU is a strongly universal family of hash functions.*

Proof. Follows as a corollary of the next lemma. □

Lemma 1. *Let $H = \{h_x : D \longrightarrow R | x \in K\}$, where R is an Abelian group and K is the set of keys, be a Δ-universal family of hash functions. Then $H' = \{h'_{x,b} : D \longrightarrow R | x \in K,\ b \in R\}$ defined by $h'_{x,b}(m) \equiv (h_x(m) + b)$ (where the addition is the operation under the group R) is a strongly universal family of hash functions.*

Proof. For all $m \neq n \in D$ and all $\alpha, \beta \in R$: $\Pr_{x,b}[h'_{x,b}(m) = \alpha,\ h'_{x,b}(n) = \beta] = \Pr_{x,b}[h_x(m) + b = \alpha,\ h_x(n) + b = \beta] = \Pr_{x,b}[h_x(m) - h_x(n) = \alpha - \beta,\ b = \alpha - h_x(m)] = \Pr_{x,b}[h_x(m) - h_x(n) = \alpha - \beta \mid b = \alpha - h_x(m)] \cdot \Pr_{x,b}[b = \alpha - h_x(m)] = 1/|R|^2$. The last equation follows since h_x is a Δ-universal hash function and $h_x(m) - h_x(n)$ takes any value in R with equal probability. □

3.3 Comparison with Linear Congruential Hash

We compare our Square Hash to the Linear Congruential Hash, which is one of the most heavily referenced Universal Hash Functions in the literature. We define the Linear Congruential Hash (LCH) family of functions to be: $LCH \equiv \{h_{x,b} : Z_p \longrightarrow Z_p | x, b \in Z_p\}$ where each of the functions $h_{x,b}$ are defined as:

$$h_{x,b}(m) \equiv mx + b \bmod p. \tag{3}$$

In most cases, the $SQHU$ family requires less computation time than LCH. The speedup occurs because squaring an n-bit number requires roughly half the number of basic word multiplications than multiplying two n-bit numbers [18]; thus we can save when dealing with quantities that are several words long. We now compare Square Hash with the MMH construction [12].

4 Comparison with MMH

Recently Halevi and Krawczyk [12] studied a family of Δ-universal hash functions entitled MMH. MMH was originally defined by Carter and Wegman. Halevi and Krawczyk discovered techniques to speed up the software implementation at negligible costs in the collision probabilities. These hash functions are suitable for very fast software implementation. They apply to hashing variable sized data and to fast cryptographic message authentication. In this section we compare our SQH family to MMH. We show that in theory SQH is more efficient with respect to both computation time and key sizes than MMH. We also show that all of the clever software optimizations discovered by Halevi and Krawczyk for MMH can be applied to SQH as well. Finally, we further optimize Square Hash by disregarding many of the carry bits in the computation. We now describe MMH^* which is the basic non-optimized version of MMH.

4.1 Description of MMH^*

Definition 6. *[12] Let $k > 0$ be an integer. Let $x = \langle x_1, \ldots, x_k \rangle$, and $m = \langle m_1, \ldots, m_k \rangle$, $x_i, m_i \in Z_p$, The MMH^* family of functions from Z_p^k to Z_p is defined as follows: $MMH^* \equiv \{g_x : Z_p^k \longrightarrow Z_p \mid x \in Z_p^k\}$ where the functions g_x are defined as*

$$g_x(m) = m \cdot x = \sum_{i=1}^{k} m_i x_i \bmod p \tag{4}$$

Theorem 3. [Halevi and Krawczyk]: *MMH^* is a Δ-universal family of hash functions.*

Halevi and Krawczyk [12] also discussed a way to generalize these functions so that their range is Z_p^l rather than just Z_p. This can be done via a Cartesian product type idea due to Stinson [27]. Specifically, we hash the message l times using l independently chosen keys and we concatenate the hashes. This yields a collision probability of $1/p^l$. At first this requires a much larger key size, but that can be reduced by applying a Toeplitz matrix type idea due to Krawczyk [16]; namely (for the case $l = 2$), choose $k + 1$ scalars $x_1, \ldots, x_{k+1}$ and set the first key to be $\langle x_1, \ldots, x_k \rangle$ and the second key to be $\langle x_2, \ldots, x_{k+1} \rangle$. The collision probability reduces to $1/p^2$.

4.2 A Variant of Square Hash Similar to MMH^*

Definition 7. *Let $k > 0$ be an integer. Let $x = \langle x_1, \ldots, x_k \rangle$, and $m = \langle m_1, \ldots, m_k \rangle$, $x_i, m_i \in Z_p$. The SQH^* family of functions from Z_p^k to Z_p is defined as follows: $SQH^* \equiv \{g_x : Z_p^k \longrightarrow Z_p \mid x \in Z_p^k\}$ where the functions g_x are defined as*

$$g_x(m) = \sum_{i=1}^{k} (m_i + x_i)^2 \bmod p \tag{5}$$

Theorem 4. *SQH^* is a Δ-universal family of hash functions.*

Proof. Let $m \neq n \in Z_p^k$ with $m = \langle m_1, \ldots, m_k \rangle$, $n = \langle n_1, \ldots, n_k \rangle$ and $m_i, n_i \in Z_p$. Let $\delta \in Z_p$. Since $m \neq n$ there is some i for which $m_i \neq n_i$. WLOG, suppose $m_1 \neq n_1$. Now, we show: $\forall x_2, \ldots, x_k \Pr_{x_1}[g_x(m) - g_x(n) = \delta] = 1/p$ (where $x = \langle x_1, \ldots, x_k \rangle$) which implies the lemma. So, $\Pr_{x_1}[g_x(m) - g_x(n) = \delta]$ $= \Pr[\sum_{i=1}^k (x_i + m_i)^2 - \sum_{i=1}^k (x_i + n_i)^2 = \delta] = \Pr[2(m_1 - n_1)x_1 = \delta - m_1^2 + n_1^2 - \sum_{i=2}^k (x_i + m_i)^2 + \sum_{i=2}^k (x_i + n_i)^2] = 1/p$. The last equation follows since $(m_1 - n_1) \neq 0$ implies that there is a unique $x_1 \in Z_p$ satisfying the equation inside the probability. □

4.3 Comparing SQH^* to MMH^*

SQH^* should be faster than MMH^* because squaring can be implemented so it requires roughly half the number of basic word multiplications as multiplying two numbers [18]. Since multiplication is relatively expensive on many architectures, we may save considerably. Halevi and Krawczyk made several clever software optimizations on MMH; the same optimizations apply to SQH as well.

4.4 Speeding up MMH^*

Here is the definition of MMH_{32}, an optimized version of MMH^*, which appeared in [12]:

Definition 8. *Set $p = 2^{32} + 15$ and $k = 32$. Let $x = \langle x_1, \ldots, x_k \rangle$, and $m = \langle m_1, \ldots, m_k \rangle$, $x_i, m_i \in Z_p$. Define the MMH_{32} family of functions from $(\{0,1\}^{32})^k$ to $\{0,1\}^{32}$ as: $MMH_{32} \equiv \{h_x : (\{0,1\}^{32})^k \longrightarrow \{0,1\}^{32} \mid x \in (\{0,1\}^{32})^k$ where the functions h_x are defined as*

$$h_x(m) = (((\sum_{i=1}^k m_i x_i) \bmod 2^{64}) \bmod (2^{32} + 15)) \bmod 2^{32} \tag{6}$$

Theorem 5. [Halevi and Krawczyk]: *MMH_{32} is an ϵ-Almost-Δ-Universal family of hash functions with $\epsilon \leq 6 \cdot 2^{-32}$.*

The same optimization applies to SQH^*.

4.5 Speeding up SQH^*

Here is a variant of Square Hash, called SQH_{asm}, which is suited for assembly language implementation.

Definition 9. *Let l and k be positive integers, and let $2^l < p < 2^l + 2^{l-1}$. Let $x = \langle x_1, \ldots, x_k \rangle$, and $m = \langle m_1, \ldots, m_k \rangle$, $x_i, m_i \in Z_p$. The SQH_{asm} family of functions from Z_p^k to $\{0,1\}^l$ is defined as follows: $SQH_{asm} \equiv \{g_x : Z_p^k \longrightarrow \{0,1\}^l \mid x \in Z_p^k\}$ where the functions g_x are defined as*

$$g_x(m) = ((\sum_{i=1}^k (m_i + x_i)^2) \bmod p) \bmod 2^l \tag{7}$$

Theorem 6. *SQH_{asm} is an ϵ-almost-Δ-universal family of hash functions with $\epsilon \leq 3 \cdot 2^{-l}$.*

Proof. Let $\delta \in \{0,1\}^l$ be chosen arbitrarily. Let $m \neq n$ be arbitrary message vectors. Let x be the key such that $h_x(m) - h_x(n) \equiv \delta \pmod{2^l}$, where $h \in SQH^*$. Equivalently, $h'_x(m) - h'_x(n) \equiv \delta \pmod{2^l}$ where $h' \in SQH_{asm}$. Now, both $h_x(m)$ and $h_x(n)$ are in the range $0, \ldots, p-1$. Therefore, their difference taken over the integers lies in the range $-p+1, \ldots, p-1$. If we denote $p = 2^l + t$ where $0 < t < 2^{l-1}$ then:

$$h'_x(m) - h'_x(n) \in \begin{cases} \{\delta, \delta - 2^l\} & t \leq \delta \leq 2^l - t \\ \{\delta - 2^l, \delta, \delta + 2^l\} & 0 \leq \delta \leq t - 1 \\ \{\delta, \delta - 2^l, \delta - 2^{l+1}\} & 2^l - t < \delta \leq 2^l - 1 \end{cases}$$

Therefore, there are at most three values for the quantity $h'_x(m) - h'_x(n)$ which cause $h_x(m) - h_x(n) \equiv v \pmod{2^l}$. Since SQH^* is a Δ-universal hash function, it follows that for any $\delta' \in \{0,1\}^l$ there is at most one choice of the key x for which $h'_x(m) - h'_x(n) \equiv \delta' \bmod p$. Therefore, at most 3 keys satisfy the equation $h_x(m) - h_x(n) \equiv \delta \pmod{2^l}$. So, $Pr_x[h_x(m) - h_x(n) \equiv \delta \pmod{2^l}] \leq 3 \cdot 2^{-l}$. □

4.6 A Further Speed-Up

There is a minor weakness in SQH_{asm}. The values m_i and x_i may each be $l+1$ bits long. We would, however, like to make l the word size of the machine on which we are implementing our code (typically $l = 32$ or 64) in order to speed up computations. Having to deal with $l+1$ bit quantities means that we have to store and square *several* extra words. A first solution is to restrict both m_i and x_i to be at most l bits. Unfortunately, $m_i + x_i$ may be an $l+1$ bit quantity which means we still need to store and square extra words. It turns out that we simply can ignore the most significant bit of $m_i + x_i$ at just a minor cost in the important statistical properties of the new hash function. We give another Square Hash variant and prove that it performs well.

Definition 10. *Let l and k be positive integers with $2^l < p < 2^l + 2^{l-1}$. Let $x = \langle x_1, \ldots, x_k \rangle$, and let $m = \langle m_1, \ldots, m_k \rangle$, $x_i, m_i \in \{0,1\}^l$. Define SQH_{asm2} family of functions from $(\{0,1\}^l)^k$ to $\{0,1\}^l$ as: $SQH_{asm2} \equiv \{g_x : (\{0,1\}^l)^k \longrightarrow \{0,1\}^l \mid x \in \{0,1\}^l\}$ where the functions g_x are defined as*

$$g_x(m) = ((\sum_{i=1}^{k}((m_i + x_i) \bmod 2^l)^2) \bmod p) \tag{8}$$

So, all we are doing is ignoring the most significant bit of $x_i + m_i$. This means that the sum will fit into l bits, which means that we do not have to use an extra word to both store and square.

Theorem 7. *SQH_{asm2} is an ϵ-almost-Δ-universal family of hash functions with $\epsilon \leq 2 \cdot 2^{-l}$.*

Proof. Let $m \neq n \in \{0,1\}^l$ with $m = \langle m_1, \ldots, m_k \rangle$, $n = \langle n_1, \ldots, n_k \rangle$ and $m_i, n_i \in \{0,1\}^l$. Let $\delta \in Z_p$. Since $m \neq n$ there is some i for which $m_i \neq n_i$. WLOG, suppose $m_1 \neq n_1$. Now, we show for all $x_2, \ldots, x_k$ $\Pr_{x_1}[g_x(m) - g_x(n) = \delta] \leq 2/2^l$ (where $x = \langle x_1, \ldots, x_k \rangle$) which implies the lemma. First, let

$$A = \sum_{i=2}^{k} ((x_i + m_i) \bmod 2^l)^2 \bmod p, \text{ and let } B = \sum_{i=2}^{k} ((x_i + n_i) \bmod 2^l)^2 \bmod p.$$

Then, $\Pr_{x_1}[g_x(m) - g_x(n) = \delta] = \Pr_{x_1}[(((x_1 + m_1) \bmod 2^l)^2 + A) - (((x_1 + n_1) \bmod 2^l)^2 + B) \equiv \delta \pmod p] = \Pr_{x_1}[((x_1 + m_1) \bmod 2^l)^2 - ((x_1 + n_1) \bmod 2^l)^2 \equiv B - A + \delta \pmod p]$. Since x_1 and m_1 are both l bit quantities, their sum will be at most $2^{l+1} - 2$, which means that to reduce this quantity mod 2^l we have to subtract off at most 2^l. Therefore,

$$((x_1 + m_1) \bmod 2^l) = x_1 + m_1 - 2^l c(x_1, m_1), \tag{9}$$

where $c(x_1, m_1)$ is some value in $\{0,1\}$. In this case, c is the carry bit associated with adding x_1 and m_1. Similarly, we can write $((x_1 + n_1) \bmod 2^l) = x_1 + n_1 - 2^l c(x_1, n_1)$. Replacing these equations into the above and performing some arithmetic manipulation, we get:

$$\begin{aligned} &\Pr_{x_1}[((x_1 + m_1) \bmod 2^l)^2 - ((x_1 + n_1) \bmod 2^l)^2 \equiv B - A + \delta \pmod p] \\ &\quad = \Pr_{x_1}[2x_1((m_1 - n_1) + (c(x_1, n_1) - c(x_1, m_1))2^l) \equiv \delta' \pmod p]. \end{aligned}$$

Where

$$\delta' = B - A + \delta + (n_1 - c(x_1, n_1)2^l)^2 - (m_1 - c(x_1, m_1)2^l)^2. \tag{10}$$

Now, $(c(x_1, n_1) - c(x_1, m_1))2^l \in \{-2^l, 0, 2^l\}$. However, since $m_1, n_1 \in \{0,1\}^l$ and since $m_1 \neq n_1$, it follows that $(m_1 - n_1) \in \{(1 - 2^l), \ldots, -1, 1, \ldots, (2^l - 1)\}$ which implies that $((m_1 - n_1) + (c(x_1, n_1) - c(x_1, m_1))2^l) \neq 0$, and for a given $c(x_1, m_1)$ and $c(x_1, n_1)$ there is at most one value of x_1 satisfying the above equations. Finally, we have

$$\begin{aligned} &\Pr_{x_1}[2x_1((m_1 - n_1) + (c(x_1, n_1) - c(x_1, m_1))2^l) \equiv \delta'] \\ &\leq \Pr_{x_1}[2x_1((m_1 - n_1) - 2^l) \equiv \delta' \pmod p | c(x_1, n_1) - c(x_1, m_1) = -1] \\ &\quad + \Pr_{x_1}[2x_1(m_1 - n_1) \equiv \delta' \pmod p | c(x_1, n_1) - c(x_1, m_1) = 0] \\ &\quad + \Pr_{x_1}[2x_1((m_1 - n_1) + 2^l) \equiv \delta' \pmod p | c(x_1, n_1) - c(x_1, m_1) = 1] \leq 3/2^l. \end{aligned}$$

This gives us a bound of $3/2^l$. We can improve this to $2/2^l$ by observing that for fixed values of m and n, $c(x_1, n_1) - c(x_1, m_1)$ cannot simultaneously take on the values $+1$ and -1 for varying choices of x_1. In particular, if $n_1 > m_1$ then we claim that $c(x_1, n_1) - c(x_1, m_1) \geq 0$. This follows because $c(x_1, n_1) - c(x_1, m_1) = -1$ implies $x_1 + n_1 < 2^l$ and $x_1 + m_1 \geq 2^l$ which implies that $m_1 > n_1$. Similarly, it can be shown that $n_1 < m_1$ implies $c(x_1, n_1) - c(x_1, m_1) \leq 0$. Thus $c(x_1, n_1) - c(x_1, m_1)$ takes on at most two possible values and ϵ is bounded by $2/2^l$. □

4.7 Ignoring Carry Bits in the Computation

We now describe a way to further speed up Square Hash at a small tradeoff in the collision probability. The idea is novel, and can be applied not only to MMH but perhaps to other constructions of universal hash functions. Basically, we show that we can ignore many of the carry bits in the computation of Square Hash and still get very strong performance for cryptographic applications. In some sense this extends the ideas of Halevi and Krawczyk who sped up MMH by ignoring only the most significant carry bit. We start by describing the notion of carry bits and explain why computation can speed up if you ignore them. We then define variants of Square Hash in which you can ignore some of the carry bits, and show that the resulting performance is still excellent for cryptographic applications. Finally, we define yet another variant of Square Hash in which you can ignore even more carry bits and show that the performance is still strong for cryptographic applications under suitable settings for the parameters.

Carry Bits When two words are added, there is usually an overflow or carry that takes place in the computation. For example, if the word size is 8, and you compute 11001001 + 10010001 you get 101011010. Since the word size is 8, the most significant bit 1 is called the carry or overflow bit because it overflowed from the usual 8 bit word size. Now, when arithmetic operations are performed on very long integers, as is usually the case for cryptographic applications, the carry bits between individual word operations are used for the next operation. So, if the word size is 8, and you are trying to compute 1011010100110101 + 1010101111100101 then the computation is broken up into parts. First, each bit string is broken up to take word size into account. The first string is broken up into two parts which we label A and B respectively: $A = 10110101$ and $B = 00110101$. The second string would be broken up into two parts: $C = 10101011$ and $D = 11100101$. Now, the computation is carried out as follows: first we compute $B + D$ store the answer in a word, and store the carry c_0 separately. Denote by E the word in which we store $B + D$. Then $E = 00011010$ and the carry bit $c_0 = 1$. Now, we compute $F = A + C + c_0$ and store the carry bit in c_1. Then $F = 01100001$ and the carry bit c_1 is 1. The total answer is the concatenation c_1FE: 10110000100011010. So, it is necessary to keep track of a carry bit as you do the computations on integers that require more than one word to represent. Unfortunately, certain instructions on processors do not deal with carry bits effectively (for example the Multiply with Accumulate instruction on the ARM). Also, even if an instruction saves the carry bit, this information may get destroyed when other instructions are executed. In addition, most high level programming languages do not deal with carry bits effectively; this increases the computation time of arithmetic instructions over integers that are several words long because it becomes necessary to explicitly keep track of the carry bit. High level programming languages are, however, preferable because they are portable and they facilitate the task of programming. We show that we can overcome these dilemmas by ignoring the carry bits altogether. We call these variants of Square Hash SQH_c and SQH_{c2} since they can be effectively implemented with high level programming languages such as C. We can prove that we get strong performance despite ignoring what seems to be a crucial part of the computation.

Ignoring Carry Bits in the Outer Summation We describe a preliminary speedup in which we ignore the carry bits in the outer summation, and show that we still get a powerful approximation to a Δ-universal hash. Let us denote by $\mathcal{C}(\Sigma_{i=1}^{n} a_i)$ the value you get if you compute the sum $\Sigma_{i=1}^{n} a_i$ but *ignore* the carry bits between the words. For example, if you let the word size be 8 and compute $a_1 = 1011010100110101$ and $a_2 = 1010101111100101$ as in the above example, then $\mathcal{C}(\Sigma_{i=1}^{2} a_i) = 0110000000011010$. The normal sum $a_1 + a_2$ is 10110000100011010. But recall that the 9th least significant bit is the result of the carry from the summation of first pair of words, and the most significant bit is the result of the carry from the second pair of words. Since you are ignoring the carry bits, the function $\mathcal{C}(\Sigma_{i=1}^{n} a_i)$ can be implemented much more efficiently than just the normal sum $\Sigma_{i=1}^{n} a_i$. This is especially true if the a_i are large integers and each require several words in order to store. We now formally define a new variant of Square Hash and show that it still gives us strong performance.

Definition 11. *Let l and k be positive integers with $2^l < p < 2^l + 2^{l-1}$. Let $x = \langle x_1, \ldots, x_k \rangle$, and $m = \langle m_1, \ldots, m_k \rangle$, $x_i, m_i \in Z_p$. The SQH_c family of functions from Z_p^k to Z_p is defined as follows: $SQH_c \equiv \{g_x : Z_p^k \longrightarrow Z_p \mid x \in Z_p^k\}$ where the functions g_x are defined as*

$$g_x(m) = (\mathcal{C}(\sum_{i=1}^{k} (m_i + x_i)^2) \bmod p) \tag{11}$$

Theorem 8. *Let l be the word size of the architecture on which you are computing and let w be the number of words it takes to store x_i. Then SQH_c is an ϵ-almost-Δ-universal family of hash functions with $\epsilon \leq 3^{2w}/2^{lw}$.*

Proof. Fix a value $a \in Z_p$ and let $m = \langle m_1, \ldots, m_k \rangle \neq m' = \langle m'_1, \ldots, m'_k \rangle$ be your two messages. Assume wlog that $m_1 \neq m'_1$. We prove that for any choice of $x_2, \ldots, x_k$ $\Pr_{x_1}[g_x(m) - g_x(m') = a \bmod p] \leq 3^{2w}/2^{lw}$ (where $x = \langle x_1, \ldots, x_k \rangle$) which implies the theorem. Now, let us fix some choice of $x_2, \ldots, x_k$ and let $s = \mathcal{C}(\sum_{i=2}^{k} (m_i + x_i)^2)$. Then,

$$\mathcal{C}(\sum_{i=1}^{k} (m_i + x_i)^2) = (x_1 + m_1)^2 + s - c \tag{12}$$

where $c \in \{0,1\}^{2l+1}$ is a "correction vector" in which the *ith* bit of c (counting from the left) contains a 1 if there was an overflow of 1 at that position (and contains a 0 otherwise). In the example above with a_1 and a_2 the correction vector c is: 1000000100000000.
Similarly, if we let $s' = \mathcal{C}(\sum_{i=2}^{k} (m'_i + x_i)^2)$ then

$$\mathcal{C}(\sum_{i=1}^{k} (m'_i + x_i)^2) = (x_1 + m'_1)^2 + s' - c' \tag{13}$$

where c' is the associated correction vector. Therefore,

$$\begin{aligned} &\Pr_{x_1}[g_x(m) - g_x(m') \equiv a \pmod p)] \\ &= \Pr_{x_1}[(x_1 + m_1)^2 + s - c - (x_1 + m_1')^2 - s' + c' \equiv a \pmod p)] \\ &= \Pr_{x_1}[x_1 \equiv (a + (c - c') + s' - s + m_1'^2 - m_1^2)/2(m_1 - m_1') \pmod p)] \\ &\leq (\text{The number of distinct values } c - c' \text{ can take}) \cdot 2^{-lw}. \end{aligned}$$

So we must derive a bound for the number of distinct values $c - c'$ can take. Now, c and c' consist mostly of 0's. In fact, the only positions of c in which there could be a 1 are the ones where there could be a carry – and since carries only occur at the boundaries between words, only bits $l+1, 2l+1, 3l+1, \ldots, 2wl+1$ can possibly contain a 1. The same hold true for c'. In the $il + 1$ bit $c_{il+1} - c'_{il+1} \in \{-1, 0, 1\}$ for $1 \leq i \leq 2l$. Since there are only $2w$ bits that can get affected and 3 different values for their difference, the total number of different vectors $c - c'$ is bounded by 3^{2w}. So, we have that $\Pr_{x_1}[g_x(m) - g_x(m') \equiv a (\text{mod} p)] \leq 3^{2w}/2^{lw}$ – which proves the theorem. □

Now, observe that the quantity $3^{2w}/2^{lw}$ is actually rather small. We see this if we substitute suitable values for the parameters. If the word size l is 32 bits, then a *computationally unbounded* adversary can forge a MAC tag of size 2,3,4, or 5 words with probability at most 2^{-57}, 2^{-86}, 2^{-115}, and 2^{-144} respectively. These are negligible and are smaller that what one may need for a reasonably secure MAC. This leads to the question of whether we can optimize further at a slightly greater cost in security. The next section works towards this aim by showing that we can ignore even more carry bits at an increased cost in collision probability.

Ignoring Carry Bits When Squaring Since the process of squaring can be expressed entirely in terms of doing basic word multiplications, shifts, and add operations, we can consider the idea of ignoring the carry bits when performing a squaring operation to further speed up our hash functions. We show that if we also ignore the carry bits that occur when the quantity $(x_i + m_i)$ is squared, then the resulting function still yields a close approximation to a δ-universal hash for suitable values for the parameters. So let's denote by $\mathcal{C}_2(a_i^2)$ the quantity obtained if you ignore the carry bits in the computation of a_i^2.

Definition 12. *Let l and k be positive integers, with $2^l < p < 2^l + 2^{l-1}$. Let $x = \langle x_1, \ldots, x_k \rangle$, and $m = \langle m_1, \ldots, m_k \rangle$, $x_i, m_i \in \{0,1\}^l$. The SQH_{c2} family of functions from $(\{0,1\}^l)^k$ to Z_p is defined as follows: $SQH_{c2} \equiv \{g_x : (\{0,1\}^l)^k \longrightarrow Z_p \mid x \in (\{0,1\}^l)^k\}$ where the functions g_x are defined as*

$$g_x(m) = (\mathcal{C}(\sum_{i=1}^{k} \mathcal{C}_2((m_i + x_i)^2)) \bmod p). \tag{14}$$

Note that we ignore carry bits when we square and when we take the sum over the $(x_i + m_i)^2$. However, we *do not* ignore the carry bits when we actually compute $(x_i + m_i)$. It is possible that we may be able to get away with ignoring these carry bits as well. We now state our main theorem about how well this new family of hash functions performs:

Theorem 9. *Let l be the word size of the architecture on which you are computing and let w be the number of words it takes to store x_i. Then SQH_{c2} is an ϵ-almost-Δ-universal family of hash functions with $\epsilon \leq (\prod_{i=1}^{w}(4i+1)^2)/2^{lw}$.*

Proof. (sketch) The proof uses similar ideas to the proof of the previous theorem about ignoring carry bits in the outer summation. In particular, we define correction vectors c and c' in the same manner as before, and bound the number of distinct possibilities for their difference $c - c'$. We first observe that for ith word of c (counting from the right for $1 \leq i \leq w$) there are 1's in at most the $\log(2i+1)$ least significant bit positions of that word – the remaining bits must be 0. So, only the least significant $\log(2i+1)$ bits in word i are undetermined. Similarly, for word j with $w+1 \leq j \leq 2w$, the least significant $\log(4w+3-2j)$ are undetermined. Moreover, we can show that if the b least significant bits of each of two different words are undetermined, then the difference of those two words can take on at most $2^{b+1} - 1$ different values. The number of distinct possible values for $c - c'$ is the product of the number of different possible values each of the individual words can take. This equals: $\prod_{i=1}^{w}(2^{\log(2i+1)+1} - 1) \cdot \prod_{j=m+1}^{2w}(2^{\log(4w+3-2j)+1} - 1)$ $= (\prod_{i=1}^{w} 2^{\log(2i+1)+1} - 1)^2 = (\prod_{i=1}^{w} 4i+1)^2$. □

Although this expression looks large, for suitable values of the parameters it still gives good security. Keep in mind that typically $1 \leq w \leq 5$. In particular, if the word size l is 32 bits, and we hash down to 2,3,4, or 5 words, then *computationally unbounded* adversaries will fail to forge the MAC with probability better than 2^{-53}, 2^{-77}, 2^{-101}, or 2^{-124} respectively.

4.8 Fully Optimized Square Hash

We present the fully optimized version of Square Hash:

Definition 13. *Let l and k be positive integers with $2^l < p < 2^l + 2^{l-1}$. Let $x = \langle x_1, \ldots, x_k \rangle$, and $m = \langle m_1, \ldots, m_k \rangle$, $x_i, m_i \in \{0,1\}^l$. The SQH_E family of functions from $(\{0,1\}^l)^k$ to $\{0,1\}^l$ is defined as follows: $SQH_E \equiv \{g_x : (\{0,1\}^l)^k \longrightarrow \{0,1\}^l \mid x \in \{0,1\}^l\}$ where the functions g_x are defined as*

$$g_x(m) = (\mathcal{C}(\sum_{i=1}^{k} \mathcal{C}_2(((m_i + x_i) \bmod 2^l)^2)) \bmod p) \bmod 2^l \tag{15}$$

Theorem 10. *Let l be the word size on which you are computing and w is the total number of words needed to store x_i. Then SQH_E is an ϵ-almost-Δ-universal family of hash functions with $\epsilon \leq (6 \cdot \prod_{i=1}^{w}(4i+1)^2)/2^{lw}$*

Proof. Combine the proofs and statements of previous theorems. □

4.9 Comparison to NMH

At the end of their paper, Halevi and Krawczyk [12] briefly discussed another family of Δ-universal hash functions called NMH. It would be interesting to do a detailed comparison between NMH and SQH that studies speed and required key sizes. Another interesting area for future research would be to apply some of our techniques of ignoring carry bits to MMH and NMH.

5 Considerations on Implementing Square Hash

In this section, we discuss various important implementation considerations, and in the next we give actual implementation results. To start with, Square Hash should be faster since we use squaring instead of multiplication. The speed-up factor for squaring an n word integer versus multiplying two n word integers is $(n-1)/2n$. Typically, MACs have tag sizes between 32 and 160 bits, depending on the level of security needed. Therefore, on 32-bit architectures, $1 \leq n \leq 5$ and we get speed up factors of %0, %25, %33, %38, and %40 for the different values of n. Now, on most slower architectures, multiplications require many more clock cycles than other simple arithmetic operations such as addition. For example, on the original Pentium processor, the ratio between number of clock cycles for unsigned multiplication versus addition is about 5:1. This ratio probably gets much larger on weaker processors such as those on cellular phones, embedded devices, smart-cards, etc,. Moreover, for these types of smaller processors, word sizes may be smaller, hence the number of words we multiply increases, and the savings we achieve by using squaring rather than multiplication greatly increases. Thus, we recommend using Square Hash on such architectures. On some of the more modern processors such as the Pentium Pro and Pentium II, multiplications do not take much more time than additions (closer to 2:1, [8]), so Square Hash is not advantageous is such cases.

Another important implementation consideration is the memory architecture of the processor on which you are implementing. In our case, we need extra data registers to quickly implement squaring. On Pentium architectures there are only 4 32-bit data registers [8]. Hence, we may need to make additional memory references which could slow things down. On the other hand, the PowerPC has 32 32-bit general purpose registers [1], which allows us to get fast squaring.

6 Implementation Results

We used the ARM (i.e. ARM7) processor to create hand optimized assembly code to compare speeds of various algorithms. The ARM7 is a popular RISC processor and is used inside cellular phones, PDAs, smartcards, etc. It is a 32 bit processor with 16 general purpose registers. Basic operations like addition require 1 cycle whereas multiplication usually requires 6 cycles.

Our results show a significant speedup for square hash over MMH, and thus validate our theoretical results. For long messages and same or better security than MMH, square hash is 1.31 times faster than MMH (Table 1).

	MMH	*SQH1*	*SQH2 (some carries dropped)*	*HMAC-SHA1*
Cycles	2061	1659	1575	4000+
Speedup over MMH	1x	1.24x	1.31x	.52x
Speedup over SHA1	1.94x	2.41x	2.54x	1x
Security - ϵ	$\frac{6.25}{2^{90}}$	$\frac{6}{2^{96}}$	$\frac{2.53}{2^{90}}$	
Code Size (bytes)	408	3040	2704	4000+
Hash key Size (random bits)	2208 (2112+96)	2112	2112	

Table 1. Assembly implementations on ARM7 with 96 bit output and input block size of 2112 bits.

Message authentication using universal hash functions is performed by breaking up a long message (e.g 1Mbyte) in to smaller blocks (e.g. 2112 bits) and reducing each block, using an equivalent size hash key, down to the size of the MAC output or tag size (e.g. 96 bits). This is repeated via a tree hash until the final tag is output. Tree hash adds about 10% overhead to both square hash and MMH [12] and we omit it in the calculations presented in the tables for purposes of simplifying comparison. The security parameter ϵ as reported in the tables would have to be multiplied by the height of the tree [12].

We report results for a tag size of 96 bits since we believe it is a popular choice for message authentication in Internet standards (e.g. HMAC). Larger output sizes of 128 and 160 bits could further improve speedup factors due to greater savings on multiplications. We also report cycle counts for SHA1 on an ARM7 to verify that we are faster than traditional non-universal hash based MACs (e.g. HMAC). To create the MAC, in actual use, MMH and square hash would have to encrypt the 96 bit output and HMAC-SHA1 would need to perform a further SHA1. We exclude this in the cycle counts in the tables to simplify comparison.

First for 2112-bit blocks (a multiple of 96) we compare MMH, $SQH1$, $SQH2$, and HMAC-SHA1. $SQH1$ is the basic square hash function SQH_{asm} with the minor optimization of SQH_{asm2} giving an overall security of $\frac{6}{2^{96}}$ compared to the security of $\frac{6.25}{2^{90}}$ for 96 bit MMH. $SQH2$ is the same as $SQH1$, except that some carry bits are dropped in the squaring until the security is similar or better than that of MMH. As a result of dropping some carries, computation time decreases.

SHA1 requires more than 1000 operations on 512-bit input and thus requires more than 4000 operations on 2112 bit input. All 3 universal hashes are significantly faster than the SHA1 based MAC. $SQH1$ is 1.24 times as fast as MMH and $SQH2$ is 1.31 times as fast as MMH. Code sizes are somewhat large because of loop unrolling. Without unrolling additional computational time will be added to all three universal hashes to handle looping. The hash key (random bits) for MMH is 96 bits larger than square hash if the Toeplitz construction is used [12].

	MMH	*SQH1*	*SQH2 (some carries dropped)*	*HMAC-SHA1*
Cycles	1086	856	816	2000
Speedup over MMH	1x	1.27x	1.33x	.54x
Speedup over SHA1	1.84x	2.34x	2.45x	1x
Code Size	220	1544	1384	4000+
Hash key Size (random bits)	1152 (1056+96)	1056	1056	

Table 2. Assembly implementations on ARM7 with 96 bit output and input block size of 1056 bits.

In Table 2 we also report cycle counts for 1056-bit blocks. Since 1024 bit blocks, as used by [12], are not a multiple of 96, we used 1056 (a multiple of 32 and 96) as the input length. We ran experiments with messages that had the same size as the tag, and we noticed similar speedups. We also tested C versions of MMH and square hash and we saw similar speedups.

Table 3 gives break downs of the instruction and cycle counts for both MMH and $SQH2$. In the future we hope to experiment with other processors.

	Instructions	*Cycles*
MMH 66 (2112/32) words	105	687
key + message loading	28	188
Multiply + Accumulate	66	461
mod p reduction	5	9
function call	6	29
Total (3 times MMH 66 words)	105	2061
SQH2 3 words	29	69
key + message loading	2	10
Multiply	5	30
Multiply + Accumulate	1	7
Adds	21	21
22 (2112/96) times SQH2 (3 words)	638	1518
mod p reduction	16	25
function overhead	10	32
Total	664	1575

Table 3. MMH and SQH2 cycle count break downs: 96 bit output and block size of 2112 bits.

7 Conclusion

We described a new family of universal hash functions geared towards high speed message authentication. On some platforms, our hash functions appear to be faster than the MMH family, which itself is considered to be one of the fastest universal hash function implementations. We also introduced additional techniques for speeding up our constructions. These constructions and techniques are general and may be of independent interest.

Acknowledgments

The authors would like to thank Shai Halevi, Dan Heer, Umair Patel, Dana Richardson, and Ganesh Sundaram for their help. We also thank the anonymous referees and program committee for their comments and suggestions.

References

1. *Rhapsody Assembler Reference.* Available via `http://developer.apple.com/`.
2. M. Atici and D. Stinson. Universal hashing and multiple authentication. In *Proc. CRYPTO 96*, Lecture Notes in Computer Science. Springer-Verlag, 1996.
3. M. Bellare, R. Canetti, , and H. Krawczyk. Keying hash functions for message authentication. In *Proc. CRYPTO 96*, Lecture Notes in Computer Science. Springer-Verlag, 1996.
4. M. Bellare, J. Killian, and P. Rogaway. The security of cipher block chaining. In *Proc. CRYPTO 94*, Lecture Notes in Computer Science. Springer-Verlag, 1994.
5. A. Bosselaers, R. Govaerts, and J. Vandewalle. Fast hashing on the Pentium. In *Proc. CRYPTO 96*, Lecture Notes in Computer Science. Springer-Verlag, 1996.
6. G. Brassard. On computationally secure authentication tags requiring short secret shared keys. In *Proc. CRYPTO 82*, Lecture Notes in Computer Science, pages 79–86, 1982.
7. L. Carter and M. Wegman. Universal hash functions. *Journal of Computer and System Sciences*, 18:143–144, 1979.
8. S. P. Dandamudi. *Introduction to Assembly Language Programming From 8086 to Pentium Proecessors.* Springer-Verlag New York, 1998.
9. E. Gilbert, F.M. Williams, and N. Sloane. Codes which detect deception. *Bell System Technical Journal*, 53(3):405–424, 1974.
10. O. Goldreich, S. Goldwasser, and S. Micali. How to construct random functions. *Journal of the ACM*, 33(4):210–217, 1986.
11. O. Goldwasser, S. Micali, and R. Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *Siam Journal of Computing*, 17(2):281–308, 1988.
12. S. Halevi and H. Krawczyk. MMH: Message authentication in software in the gbit/second rates. In *Proceedings of the 4th Workshop on Fast Software Encryption*, Lecture Notes in Computer Science. Springer-Verlag, 1997.
13. T. Helleseth and T. Johansson. Universal hash functions from exponential sums over finite fields. In *Proc. CRYPTO 96*, Lecture Notes in Computer Science. Springer-Verlag, 1996.
14. T. Johansson. Bucket hashing with small key size. In *Proc. EUROCRYPT 97*, Lecture Notes in Computer Science. Springer-Verlag, 1997.

15. L. Knudsen. Truncated and higher order differentials. In *Proceedings of the 2nd Workshop on Fast Software Encryption*, Lecture Notes in Computer Science. Springer-Verlag, 1995.
16. H. Krawczyk. LFSR-based hashing and authentication. In *Proc. CRYPTO 94*, Lecture Notes in Computer Science. Springer-Verlag, 1994.
17. H. Krawczyk. New hash functions for message authentication. In *Proc. EUROCRYPT 95*, Lecture Notes in Computer Science. Springer-Verlag, 1995.
18. A.J. Menezes, P.C. van Oorschot, and S.A. Vanstone. *Handbook of Applied Cryptography.* CRC Press, 1997.
19. M. Naor and O. Reingold. On the construction of pseudo-random permutations: Luby-rackoff revisited. *Journal of Cryptology*, 1999. To Appear. Extended abstract in: Proc. 29th Ann. ACM Symp. on Theory of Computing, 1997, pp. 189-199.
20. W. Nevelsteen and B. Preneel. Software performance of universal hash functions. In *Proc. EUROCRYPT 99*, Lecture Notes in Computer Science. Springer-Verlag, 1999.
21. National Bureau of Standards. FIPS publication 46: Data encryption standard, 1977. Federal Information Processing Standards Publication 46.
22. S. Patel, Z.A. Ramzan, and G. Sundaram. Towards making Luby-Rackoff ciphers practical and optimal. In *Proceedings of Sixth Workshop on Fast Software Encryption*, March 1999. To Appear.
23. B. Preneel and P.C. van Oorschot. On the security of two MAC algorithms. In *Proc. EUROCRYPT 96*, Lecture Notes in Computer Science. Springer-Verlag, 1996.
24. R. Rivest. The MD5 message digest algorithm. IETF RFC-1321, 1992.
25. P. Rogaway. Bucket hashing and its application to fast message authentication. In *Proc. CRYPTO 95*, Lecture Notes in Computer Science. Springer-Verlag, 1995.
26. V. Shoup. On fast and provably secure message authentication based on universal hashing. In *Proc. CRYPTO 96*, Lecture Notes in Computer Science. Springer-Verlag, 1996.
27. D. Stinson. Universal hashing and authentication codes. *Designs, Codes, and Cryptography*, 4:369–380, 1994.
28. M. Wegman and L. Carter. New hash functions and their use in authentication and set equality. *Journal of Computer and System Sciences*, 22:265–279, 1981.