DAKOTA – Hashing from a Combination of Modular Arithmetic and Symmetric Cryptography

Ivan B. Damgård[1], Lars R. Knudsen[2], and Søren S. Thomsen[2]

[1] Department of Computer Science, University of Aarhus,
IT-parken, Aabogade 34, DK-8200 Aarhus N, Denmark
[2] Department of Mathematics, Technical University of Denmark,
Matematiktorvet 303S, DK-2800 Kgs. Lyngby, Denmark

Abstract. In this paper a cryptographic hash function is proposed, where collision resistance is based upon an assumption that involves squaring modulo an RSA modulus in combination with a one-way function that does not compress its input, and may therefore be constructed from standard techniques and assumptions. We are not able to reduce collision finding to factoring, but on the other hand, our hash function is more efficient than any known construction that makes use of modular squaring.

1 Introduction

With the emergence of a large number of attacks (e.g. [1, 2, 4, 11, 27, 28, 29]) on many dedicated hash functions, the development of alternatives based on different design principles has become increasingly desirable. While much research has shifted towards hash functions based on block ciphers, some recent proposals such as [5, 7, 12, 16, 17] aim for some sort of "provable security". Provable security is in quotation marks because security proofs are always based on some unproven assumption.

The downside to provably secure hash functions is that they are often much less efficient than commonly used alternatives like MD5 [24] and SHA-1 [21], and with the arguably most common application of hash functions being the pre-processing of a message in a digital signature scheme with the intention of making the entire signing process faster, a slow hash function loses some of its justification. Hence, speed cannot be overlooked altogether, but keeping in mind the large number of recent attacks on fast dedicated hash functions, it may be beneficial to reconsider the scaling of speed vs. security.

In this paper, we propose a reasonably fast hash function based partly on number theoretic principles. We prove that finding collisions is as hard as breaking a computational assumption that involves squaring modulo an RSA modulus in combination with a one-way, collision resistant function that *does not need to compress* (hence it can be injective). We propose different variants of such a function, and we urge the community to assist in analysing these variants and to possibly propose others.

S.M. Bellovin et al. (Eds.): ACNS 2008, LNCS 5037, pp. 144–155, 2008.

2 Hash Function Security

Three types of attack on hash functions are usually mentioned in the literature. These are the preimage attack, the second preimage attack and the collision attack, with expected complexities for an ideal m-bit hash function of, respectively, 2^m, 2^m, and $2^{m/2}$. In most cases these are also the conjectured complexities of a newly designed hash function, and if an attack of lower complexity is discovered, then the hash function is usually considered broken.

Finding second preimages in a hash function is at least as hard as finding collisions, since given a second preimage, one immediately has a collision. Moreover, if we let H be a hash function that accepts inputs which are (much) longer than the outputs of H, then an algorithm that finds preimages might be given the hash value $H(x)$ for some x. With good probability, the algorithm will return a preimage $\tilde{x}$ of $H(x)$ such that $x \neq \tilde{x}$. This also yields a collision. Hence, a collision resistant hash function may reasonably be considered resistant also to preimage attacks.

Hash functions aimed towards the use in applications such as digital signatures, and to some extent commitment schemes and data integrity, must be collision resistant. Hence, the complexity of finding collisions must be high enough that the task is infeasible, preferably for many years to come. For such hash functions it makes sense for the designers to indicate one claimed attack complexity, a complexity that is a lower bound for all three mentioned types of attack.

This is the choice we have made here. We shall not claim that our hash function behaves as a random oracle. We simply claim that it is collision intractable.

3 The Proposal

The hash function proposal of this paper is now presented. We call this hash function DAKOTA.

The basic idea can be seen as a further development of earlier hash functions whose security is provably reducible to factoring [14,9]. A representative example of these ideas is where the compression function h maps as $h : \{0,1\} \times SQ(n) \to SQ(n)$, and is defined by $h(b, y) = a_b y^2 \bmod n$, where a_0, a_1 are randomly chosen squares modulo RSA modulus n, and $SQ(n)$ is the set of squares modulo n. It is easy to show that an algorithm that finds collisions for h can be used to factor n with probability $1/2$. The actual hash function H is obtained by iterating h in a standard Merkle-Damgård construction, using a random square y_0 as initial value.

An alternative formulation, better suited for generalisation is to define a function $f : \{0,1\} \to SQ(n)$, where $f(b) = a_b$ and write $h(b, y) = f(b)y^2 \bmod n$. The assumption that h is collision intractable can be phrased as saying that it is hard to find distinct inputs $(b, y), (b', y')$ such that $f(b)f(b')^{-1} = (y'y^{-1})^2 \bmod n$.

While this construction is very inefficient, since we spend a modular squaring to hash a single bit, efficiency can be improved by extending the input domain of f: we can define instead f to take input from $\{0,1\}^t$, and select 2^t random

squares as output values. This is t times faster than the basic idea. The reduction to factoring can be constructed to have success probability independent of t, see [9]. However, this idea is of course only practical for small values of t.

A natural alternative seems to be to define an efficient algorithm for computing f, instead of specifying it as a table with all output values. In this way we can allow f to take large strings as input and gain efficiency. The price to pay, as we shall see, is that we can no longer prove that security is equivalent to factoring – we return later to a discussion of what we can prove instead. A more immediate practical problem, however, is that the security of the original construction requires f to map into $SQ(n)$, and it is not clear how we can ensure that this happens when we have too many inputs to be able to specify the function by a table. To hit $SQ(n)$, it seems that the algorithm computing f would have to either (a) square a known value or (b) make use of the factorisation of n. In the case of (a), this would mean that the adversary would know the square root of the output of f, and this would invalidate the reduction to factoring. With respect to (b), this does not work because f has to be a public function, so it cannot be based on the secret factors of n.

We therefore propose below a variant of the idea, that permits f to map into all of $\mathbb{Z}_n$ while still allowing a security proof.

To this end, we assume we are given a probabilistic algorithm $\mathcal{G}$ that on input a security parameter k produces an RSA modulus $n = pq > 2^k$, where $p \equiv q \equiv 3 \pmod 4$ (this means that -1 is a quadratic non-residue mod n, and that squaring mod n is a bijection on the set of quadratic residues, see e.g. [18]), together with a description of a function $f : \{0,1\}^k \to \mathbb{Z}_n$. As we shall see, f will have to be one-way and collision intractable, but since f does not compress its input, there is no circularity here. Finally, it chooses $r \in \mathbb{Z}_n^*$ at random, sets $s = r^2 \bmod n$ and returns (f, n, s) (p, q, and r are discarded in a secure manner).

Using the output of $\mathcal{G}$, we define a compression function $h : \{0,1\}^k \times \mathbb{Z}_n \to \mathbb{Z}_n$ as follows:

$$h(x, y) = (f(x)y)^2 \bmod n.$$

From this compression function we can build a collision intractable hash function under the following assumption.

Assumption 1. Consider a probabilistic polynomial time algorithm that takes as input f, n as produced by $\mathcal{G}$ on input k and outputs $x, \tilde{x}, z$. Then the probability that $x \neq \tilde{x}$ and $f(x)/f(\tilde{x}) = \pm z^2 \bmod n$ is negligible.

Assumption 1 leads to some requirements on the function f. For a discussion, see Section 3.1.

We now define our hash function H by a standard Merkle-Damgård construction [10, 19]: split the input message x into blocks of length k bits, call them $x_1, \ldots, x_t$. We assume for simplicity that x has been padded to a length divisible by k. Then define $y_0 = s$, and $y_i = h(x_i, y_{i-1})$ for $1 \leq i \leq t$. Finally, set $H(x) = y_t$.

Collision intractability of H does not follow as usual with this type of iterated construction because our compression function is not collision intractable (inputs of the form $(x, y), (x, -y)$ collide). However, we can still prove:

Theorem 1. *The hash function H as described above is collision intractable under Assumption 1.*

Proof. We assume we are given an algorithm $\mathcal{A}$ that finds collisions with probability ϵ. We then build an algorithm that breaks Assumption 1: we get f, n as input, we choose r at random in $\mathbb{Z}_n^*$, set $s = r^2 \bmod n$ and we give f, n, s to $\mathcal{A}$.

Assume now that a collision has been found by $\mathcal{A}$, for different inputs $x_1, \ldots, x_t$ and $\tilde{x}_1, \ldots, \tilde{x}_{\tilde{t}}$. This gives rise to two sequences of values $y_0, \ldots, y_t$ and $\tilde{y}_0, \ldots, \tilde{y}_{\tilde{t}}$, where $y_t = \tilde{y}_{\tilde{t}}$. This implies that we have $h(x_t, y_{t-1}) = h(\tilde{x}_{\tilde{t}}, \tilde{y}_{\tilde{t}-1})$, or equivalently

$$(f(x_t)/f(\tilde{x}_{\tilde{t}}))^2 = (\tilde{y}_{\tilde{t}-1}/y_{t-1})^2 \bmod n.$$

If $f(x_t)/f(\tilde{x}_{\tilde{t}}) \neq \pm\tilde{y}_{\tilde{t}-1}/y_{t-1} \bmod n$, then we can efficiently factor n by computing (e.g.) $p = \gcd(f(x_t)/f(\tilde{x}_{\tilde{t}}) - \tilde{y}_{\tilde{t}-1}/y_{t-1}, n)$. Factoring n in particular allows breaking Assumption 1.

So we may assume that $f(x_t)/f(\tilde{x}_{\tilde{t}}) = \pm\tilde{y}_{\tilde{t}-1}/y_{t-1} \bmod n$. Now, if $x_t \neq \tilde{x}_{\tilde{t}}$, we can break Assumption 1, using the fact that we can compute a square root z of $\tilde{y}_{\tilde{t}-1}/y_{t-1} \bmod n$. This is because we already know a square root of y_{t-1}, namely $f(x_{t-1})y_{t-2} \bmod n$ (if $t > 1$) or r (if $t = 1$); and by the same argument we also know a square root of $\tilde{y}_{\tilde{t}-1}$.

On the other hand, if $x_t = \tilde{x}_{\tilde{t}}$ then $1 = f(x_t)/f(\tilde{x}_{\tilde{t}}) = \pm\tilde{y}_{\tilde{t}-1}/y_{t-1} \bmod n$. Note that $\tilde{y}_{\tilde{t}-1}/y_{t-1} \in SQ(n)$ since all y-values are squares modulo n, but $-1 \notin SQ(n)$, so we conclude that $\tilde{y}_{\tilde{t}-1}/y_{t-1} = 1 \iff \tilde{y}_{\tilde{t}-1} = y_{t-1}$, and we can now repeat the same argument.

The only way in which this can fail to produce a contradiction with Assumption 1 is if we end up concluding that $y_t = \tilde{y}_{\tilde{t}}, y_{t-1} = \tilde{y}_{\tilde{t}-1}, \ldots, y_0 = \tilde{y}_{\tilde{t}-t}$, where we assume without loss of generality that $\tilde{t} \geq t$. Since the two inputs are different it must be that $\tilde{t} > t$, whence we have that

$$s = y_0 = h(\tilde{x}_{\tilde{t}-t}, \tilde{y}_{\tilde{t}-t-1}) = (f(\tilde{x}_{\tilde{t}-t})\tilde{y}_{\tilde{t}-t-1})^2 \bmod n.$$

In other words, we have produced a square root of s. Since r was uniformly chosen, there is a probability of $1/2$ that the square root we find here is different from $\pm r \bmod n$, in which case we can factor n and break Assumption 1. Hence if $\mathcal{A}$ finds a collision with probability ϵ, we can break the assumption with probability at least $\epsilon/2$. □

Remark 1. If n can be factored, then Assumption 1 can be broken, and this may allow for an attack on the hash function itself. Hence, n must be generated in such a way that nobody knows its factorisation. Boneh and Franklin [3] have described efficient techniques to do this securely.

3.1 Notes on Assumption 1

If f can be inverted on a non-negligible subset of $\mathbb{Z}_n$, then Assumption 1 can be broken. One simply chooses $\tilde{x}$ and z, and computes $f^{-1}(z^2 f(\tilde{x}))$. In particular, it must be infeasible to find a zero of f, since otherwise Assumption 1 is broken with $z = 0$ and any $\tilde{x}$.

On the other hand, the image of f may only be a small subset of $\mathbb{Z}_n$, and in such cases the ability to find a preimage of a given image of f may not be a problem. This is due to the fact that the size of the image of the function $F(x, \tilde{x}) = f(x)/f(\tilde{x}) \bmod n$ can be made much smaller than n, for instance $n/2^{128}$. In this case, assuming that the outputs of F are uniformly distributed in $\mathbb{Z}_n$, the probability that for a given z, an input $(x, \tilde{x})$ to F *exists* such that $F(x, \tilde{x}) = \pm z^2 \bmod n$, is only about 2^{-128}. Of course, if f expands, then the efficiency of the hash function decreases.

It is clear why f must be collision intractable, as mentioned: if $f(x) = f(\tilde{x})$ for $x \neq \tilde{x}$, then Assumption 1 is broken with $z = 1$. However, this possibility is excluded if we design f to be injective.

If f is one-way, and n cannot be factorised, then attacks where one chooses two values of $(x, \tilde{x}, z)$ and computes the third will fail. Computing z for given $(x, \tilde{x})$ requires computing a square root modulo n. Computing, say, x given $(\tilde{x}, z)$, requires inverting f.

Another generic attack is to compute $z^2 \bmod n$ for many random values of z, likewise many values of $f(x)/f(\tilde{x}) \bmod n$, and hope for a collision. This is just a standard meet-in-the-middle attack which has complexity about $\sqrt{n}$ and hence has no practical significance.

Whether a more efficient method to find collisions than by factoring n is possible, depends of course on whether the design of f interacts in some unfortunate way with arithmetic modulo n. For instance, one may try to find $x, \tilde{x}$ such that $f(x) = -f(\tilde{x}) \bmod n$, or in general $x, \tilde{x}$ such that $f(x) = \pm a^2 f(\tilde{x}) \bmod n$ for some a of the adversary's choice. We return to this question below.

3.2 Output Transformation

For most applications it is recommended that the output of H is not used as the final hash, but is instead fed to an output transformation function $\Omega : \mathbb{Z}_n \rightarrow \{0,1\}^m$, where m is chosen such that the complexity of factoring n is no less than $2^{m/2}$. The intention is to obtain an output size corresponding to the security level of the hash function. In addition, Ω might be used to obfuscate the algebraic structure of the output. Any algebraic structure could compromise the security of schemes in which the hash function is used, particularly schemes that are based on modular arithmetic such as signature schemes based on RSA [25, 26]. A third purpose of an output transformation may be to improve the preimage resistance of the hash function.

In order to provably extend the collision resistance of H to $\Omega \circ H$, Ω would itself have to be collision intractable. But in practice this may not be necessary: even if it is easy to find collisions for Ω, these do not necessarily lead to collisions for $\Omega \circ H$. In fact, it may be sufficient that Ω mixes the bits of its input well such that all output bits depend on all input bits, and that it is regular meaning that all 2^m preimage sets are of roughly the same size. Unless the hash function is primarily used for short messages, Ω does not have to be terribly fast, since it is only used once.

A concrete example of how the output transformation could work is as follows: Let G be a finite group of prime order and let $g_1, ..., g_u$ be chosen randomly in G. We choose the two integers t and u such that $t < \log_2 |G|$ and $tu \geq \log_2(n)$. Let $b = H(x) = b_1 \| \cdots \| b_u$ be the output from the main hash function, where the length of each b_i is t. Since $tu \geq \log_2(n)$ this may require that b_u be padded with zeros – a simple padding scheme is fine here, since all inputs to the output transformation will have the same length. We then define the intermediate output $\omega(b)$ as $g_1^{b_1} \cdots g_u^{b_u}$, that is, a single element from G.

It is well known [6] (and straightforward to show) that finding collisions for this mapping is as hard as solving the discrete logarithm problem (DLP) in G. There are well known constructions of such groups based on elliptic curves, where the DLP can be reasonably assumed to be hard, and where the representation of a group element is 200-400 bits long. Note also that by choosing t small, for instance ≤ 8, we may perform the exponentiations $g_i^{b_i}$ by table lookups.

We recommend that a final bijective function based on symmetric cryptography is used on $\omega(b)$ to produce the final output $\Omega(b)$, in order to prevent the adversary from exploiting the algebraic properties of exponentiation in G. This function could be designed based on a block cipher in CBC mode as described in more detail in Section 4.1.

4 Proposals for f

In this section we describe a number of possible instantiations of f, the function used in the compression function of DAKOTA which must satisfy Assumption 1.

As mentioned, for f to satisfy Assumption 1, it cannot be easily invertible, and it must also be collision resistant. Hence, it might be desirable that f be injective. It should take a k-bit input and return an element of $\mathbb{Z}_n$.

One-wayness and collision resistance are necessary, but not sufficient conditions. As an example, consider $f(x) = x^2 \bmod n$, where $x > n/2$. It is generally believed that for a secure modulus n, this function is one-way and collision resistant. However, this is clearly a bad proposal, since given z and $\tilde{x}$, one may choose $x = z\tilde{x}$ and thus break Assumption 1.

If, however, one assumes that f is, in some sense, independent of arithmetic modulo n, or put differently, does not interact badly with arithmetic mod n, then, intuitively, it seems that one-wayness and collision resistance are indeed sufficient properties. This idea is made more concrete in the proposals below.

In the following we assume a modulus of size about 1024 bits. A proposal for f should be evaluated on its security properties, and also on its scalability, i.e. how easily it can be adapted to a new modulus of a different size.

4.1 Combining Modular Arithmetic with Symmetric Encryption

It is quite straightforward to construct a one-to-one function that has "nothing" to do with modular arithmetic in the sense that we make it hard for an adversary to choose inputs for which the outputs satisfy some algebraic relation mod n: one can simply encrypt the input under a fixed key using a symmetric algorithm,

say, a block cipher in CBC mode. To make life harder for the adversary, it may be desirable that all output bits depend on all input bits. This can be ensured by encrypting twice, reversing the order of the blocks for the second encryption. Unfortunately, anything based on encryption under a fixed key is easy to invert, and so this is not sufficient to get what we want.

On the other hand, if we are willing to use modular arithmetic, we can easily satisfy a different subset of our demands, namely we can build a fast one-way and one-to-one function by using modular squaring. Actually, for an RSA modulus, squaring is a four-to-one mapping, but if we constrain the input to be less than half the modulus, one can only find collisions by factoring the modulus, so this is "as good as" being one-to-one.

An obvious idea to get all the properties we want is now to combine the two ideas: first square to get the one-way property and then do AES encryption to obfuscate the algebraic structure. This results in the following proposal for f (keep in mind that k is the size (number of bits) of the input to f):

Proposal 1. We assume the size of n is 1025 bits, and we choose another RSA modulus n' of size 1024 bits. We let $k = 1022$ and x be the input, and let E_κ be AES encryption in CBC mode with key κ. κ_1 and κ_2 are two fixed AES keys.

- Let $u = x^2 \bmod n'$.
- Let $v = E_{\kappa_1}(u) = v_1 \| \cdots \| v_8$.
- Let $f(x) = E_{\kappa_2}(v_8 \| \cdots \| v_1)$ (notice that the order of the blocks of v is reversed).

The specific choice of input and output sizes is just to make sure that inputs are less than $n'/2$ and that outputs are less than n.

With this construction of f, we can hash about 128 bytes using two modular squarings, a multiplication and AES encryption of 256 bytes.

How hard is it for the adversary to break our assumption for this choice of f? The good news is that the straightforward ways to attack will not work: the adversary may compute $f(x), f(\tilde{x})$ and try to extract a square root of $\pm f(x)/f(\tilde{x}) \bmod n$, he can try to invert f or he can try to find a collision of f. All three types of attack are as hard as factoring n or n'.

However, for attacks that choose a and then try to find $x, \tilde{x}$ such that $f(x)/f(\tilde{x}) = \pm a^2 \bmod n$, we can only base ourselves on the heuristic that the design of f is sufficiently incompatible with arithmetic modulo n for this to be infeasible.

Proposal 1 is easily adapted to an RSA modulus n of a different size: simply choose n' as an RSA modulus smaller than $n/2$, and ensure $0 \leq x < n'/2$. Furthermore, the double CBC encryption will have to accommodate a different number of blocks.

4.2 A Proposal Using a k-Bit Permutation

If a (suitable) k-bit permutation g is available, then one might define f simply as $g(x) \oplus x$. This is a standard method of obtaining a one-way function from

a ("good") invertible permutation, see e.g. [19]. For $k = 1024$ we suggest the following definition of f.

Proposal 2. Define f as $g(x) \oplus x$, where g is defined as follows. Let τ be an invertible function that transforms the 1024-bit string x into an 8×8 matrix A of 16-bit values. Let A_i be row i of A, and let $\mathbf{E}$ be the function operating on A by replacing A_i with $E_i(A_i)$, $0 \leq i < 8$, where E_κ is the AES encryption function with key κ. Define a *round* as

$$A \leftarrow \mathbf{E}(A)^{\mathrm{T}}$$

(where $(\cdot)^{\mathrm{T}}$ is the transpose operator). Perform 4 such rounds. Finally, let $g(x) = \tau^{-1}(A)$.

Every state bit depends on every input bit after just two rounds. It seems very hard to identify a useful structure in g or its inverse.

Proposal 2 is easily adapted to, e.g., a 2049-bit modulus as follows: Let x be a 2048-bit input string, and form from x a 16×16 matrix of (8-bit) bytes. Define $\mathbf{E}$ as above, only with $0 \leq i < 16$, and perform again 4 rounds as defined above.

5 Performance

Our proposal may be compared to the basic version of VSH [7] (see also Section 6) as follows: in both hash functions a multiplication and a squaring modulo n must be performed for each message block, plus an overhead which in the case of VSH is due to the computation of the product of small primes, and in our case is due to the evaluation of f. Our hash function is likely to perform better for one important reason: the size of a message block in our case is up to $\log_2(n)$ bits, whereas in the case of (basic) VSH it is the largest number t such that the product of the first t primes is less than n (in [7], t is estimated to be approximately $\frac{\log n}{\log \log n}$). As an example, with n begin a 1024-bit modulus, the size of a message block in our hash function may be up to 1024 bits, and in VSH it would be 131 bits.

There may also be an important difference in efficiency between evaluating f and computing the product of up to t primes.

There are faster versions of VSH that use larger message blocks and precompute some products of the small primes. These versions require a larger modulus to be used, and reliably comparing fast VSH with DAKOTA requires implementations using similar optimisations, compilers, processors etc. According to measurements presented in [7], fast VSH with claimed security equivalent to factoring a 1024-bit modulus reaches speeds of around 840 cycles/byte (on a 1GHz Pentium III, which is a 32-bit processor).

The two described versions of DAKOTA have been implemented with random choices of the moduli n (1025 bits) and n' (1024 bits), the square s and AES keys κ_1 and κ_2. We used our own C implementation of AES, which does not achieve quite the same speeds as, e.g., those of the Crypto++ [8] library.

Table 1. Speed comparison of DAKOTA with SHA-256 (see text for details)

Hash function	Approximate speed (cycles/byte)	
	32-bit	64-bit
SHA-256	20	20
DAKOTA (Proposal 1)	385	170
DAKOTA (Proposal 2)	330	170

For large integer arithmetic, the GMP (GNU Multiple Precision) arithmetic library [13] has been used (version 4.2.2). The implementation was compiled and run in 32-bit and 64-bit modes, see Table 1 for benchmarks. In the table are also included benchmarks for the Crypto++ [8] (version 5.5) implementation of SHA-256 [22]. All benchmarks refer to a test on a 2.4GHz Intel Core 2 Duo processor. The DAKOTA implementations were compiled using `gcc` version 4.1.3 (with optimisation flags `-static -O2 -fomit-frame-pointer`) in GNU/Linux Ubuntu 7.10. Further optimisations of the implementations are almost certainly possible.

We have not performed tests using larger modulus sizes, although in order to achieve security (with respect to collisions) comparable to, e.g., SHA-256, a modulus size of about 3072 bits would be needed, according to estimates by NIST [23].

5.1 Montgomery Multiplication

The Montgomery reduction [20] is a means to speed up modular computations. Let R be an integer, which in our case could be chosen to some power of two. In a modular multiplication modulo n of integers x and y, one first computes $x' = xR \bmod n$ and $y' = yR \bmod n$. The Montgomery reduction of $x'y'$ is then $x'y'R^{-1} = xyR \bmod n$.

In our proposal we are computing a series of values of the form $y_{i+1} = (f(x_i)y_i)^2 \bmod n$. We will introduce a variant of the Montgomery reduction tailored for our computations.

Let $\tilde{y}_0 = y_0R^3 \bmod n$, then compute the Montgomery reduction of $f(x)\tilde{y}_0$ which yields $w = f(x)y_0R^2 \bmod n$, then compute the Montgomery reduction of ww, which is

$$\tilde{y}_1 = (f(x)y_0)^2R^3 \bmod n = y_1R^3 \bmod n.$$

This method can be iterated such that given $y_iR^3 \bmod n$ we get $y_{i+1}R^3 \bmod n$ using one multiplication, one squaring and two Montgomery reductions. When all message blocks have been processed in the hash function, the constant R^3 can be removed.

6 Related Work

A number of hash functions claiming to obtain provable security have been proposed in the past. Some examples are now mentioned.

Goldwasser-Micali-Rivest. Following ideas of [14], a provably collision intractable hash function can be constructed as described in Section 3 (see also [9]). For completeness, we describe the construction again here: Let a_0, a_1 be random squares modulo RSA modulus n. All values are public, but the factors of n are secret. Define the compression function $h : \{0,1\} \times \mathbb{Z}_n^* \to \mathbb{Z}_n^*$ by

$$h(x, y) = a_x y^2 \bmod n.$$

It follows that a collision gives a square root modulo n which (with probability 1/2) can be used to factor n. Variants that are t times faster have been proposed [9]; these use 2^t public squares and preserve the collision intractability.

VSH. A more recent example of a provably collision resistant hash function is VSH [7], which is roughly defined as follows. Let n be a public RSA modulus. Let $p_1, \ldots, p_k$ be public primes such that $\prod_{i=1}^{k} p_i < n$. Define the compression function $h : \{0,1\}^k \times \mathbb{Z}_n^* \to \mathbb{Z}_n^*$ by

$$h(x, y) = y^2 \prod_{i=1}^{k} p_i^{x_i} \bmod n,$$

where x_i is the ith bit of x. The security of this construction relies on the so-called Very Smooth Square Root Problem, which is connected to the difficulty of factoring.

Discrete log hash. The discrete log hash, or the Chaum-van Heijst-Pfitzmann hash function [6] is defined as follows. Let p and $q = \frac{p-1}{2}$ be large, odd primes. Let α and β be randomly chosen primitive elements of $\mathbb{Z}_p$, such that $\log_\alpha(\beta)$ is hard to find. Define the compression function $h : \mathbb{Z}_q \times \mathbb{Z}_q \to \mathbb{Z}_p^*$ by

$$h(x, y) = \alpha^x \beta^y \bmod p.$$

Then it can be shown that a collision for h enables one to compute $\log_\alpha(\beta)$.

MASH. The MASH (for Modular Arithmetic Secure Hash) functions [15] are standardised in ISO/IEC 10118-4:1998. The compression function of MASH-1 is defined as follows. Let n be an RSA modulus, and let the message block be expanded to x, where the 4 most significant bits of every byte are set to `1111` (except in the final (padding) block, where `1010` is inserted). Let $a =$ `f00...00` (in hexadecimal), and let

$$h(x, y) = \Big(((x \oplus y) \vee a)^2 \bmod n\Big) \oplus y.$$

In MASH-2, the exponent 2 is replaced by $2^8 + 1$.

The MASH functions fall somewhat outside the category of provably secure hash functions, since no security proof exists. The claimed security of both these hash functions is $n^{1/2}$ for preimages, and $n^{1/4}$ for collisions.

7 Conclusion

The DAKOTA hash function is proposed. The properties of DAKOTA are that it reduces the problem of constructing a secure (collision resistant) compression function to the problem of constructing a function f such that it is infeasible to find $x, \tilde{x}, z$ with $f(x)/f(\tilde{x}) = \pm z^2 \bmod n$, given that factoring the RSA modulus n is infeasible. The function f must be collision resistant and one-way, but it does not need to compress, and hence it can be injective.

Two proposals for the function f have been given. One combines modular arithmetic with symmetric encryption, and the other uses only symmetric encryption in the form of the AES encryption function in black-box mode.

For both versions of DAKOTA, performance is good compared to other hash functions based on modular arithmetic.

Acknowledgments

We would like to thank Dan Bernstein, Thomas Shrimpton, and the anonymous referees for helpful comments and discussions.

References

1. Biham, E., Chen, R.: Near-Collisions of SHA-0. In: Franklin, M. (ed.) CRYPTO 2004. LNCS, vol. 3152, pp. 290–305. Springer, Heidelberg (2004)
2. Biham, E., Chen, R., Joux, A., Carribault, P., Lemuet, C., Jalby, W.: Collisions of SHA-0 and Reduced SHA-1. In: Cramer, R.J.F. (ed.) EUROCRYPT 2005. LNCS, vol. 3494, pp. 36–57. Springer, Heidelberg (2005)
3. Boneh, D., Franklin, M.K.: Efficient Generation of Shared RSA Keys (Extended Abstract). In: Kaliski Jr., B.S. (ed.) CRYPTO 1997. LNCS, vol. 1294, pp. 425–439. Springer, Heidelberg (1997)
4. Chabaud, F., Joux, A.: Differential Collisions in SHA-0. In: Krawczyk, H. (ed.) CRYPTO 1998. LNCS, vol. 1462, pp. 56–71. Springer, Heidelberg (1998)
5. Charles, D., Goren, E., Lauter, K.: Cryptographic Hash Functions from Expander Graphs. In: NIST Second Cryptographic Hash Workshop, Corwin Pavilion, UCSB Santa Barbara, California, USA, August 24–25 (2006), `http://csrc.nist.gov/groups/ST/hash/documents/LAUTER_HashJuly27.pdf` [2008/1/14]
6. Chaum, D., van Heijst, E., Pfitzmann, B.: Cryptographically Strong Undeniable Signatures, Unconditionally Secure for the Signer. In: Feigenbaum, J. (ed.) CRYPTO 1991. LNCS, vol. 576, pp. 470–484. Springer, Heidelberg (1992)
7. Contini, S., Lenstra, A.K., Steinfeld, R.: VSH, an Efficient and Provable Collision-Resistant Hash Function. In: Vaudenay, S. (ed.) EUROCRYPT 2006. LNCS, vol. 4004, pp. 165–182. Springer, Heidelberg (2006)
8. Dai, W.: Crypto++® Library 5.5.2 (2007), `http://www.cryptopp.com` [2008/1/11]
9. Damgård, I.: Collision Free Hash Functions and Public Key Signature Schemes. In: Chaum, D., Price, W.L. (eds.) EUROCRYPT 1987. LNCS, vol. 304, pp. 203–216. Springer, Heidelberg (1988)

10. Damgård, I.: A Design Principle for Hash Functions. In: Brassard, G. (ed.) CRYPTO 1989. LNCS, vol. 435, pp. 416–427. Springer, Heidelberg (1990)
11. Dobbertin, H.: Cryptanalysis of MD4. In: Gollmann, D. (ed.) FSE 1996. LNCS, vol. 1039, pp. 53–69. Springer, Heidelberg (1996)
12. Finiasz, M., Gaborit, P., Sendrier, N.: Improved fast syndrome based cryptographic hash function. In: ECRYPT Hash Workshop, Barcelona, Spain, May 24–25 (2007), http://events.iaik.tugraz.at/HashWorkshop07/papers/Finiasz_ImprovedFastSyndromeBasedCryptographicHashFunction.pdf [2008/1/3]
13. The GNU MP Bignum Library (2007), http://gmplib.org [2008/3/25]
14. Goldwasser, S., Micali, S., Rivest, R.L.: A Digital Signature Scheme Secure Against Adaptive Chosen-Message Attacks. SIAM Journal on Computing 17(2), 281–308 (1988)
15. ISO/IEC 10118-4:1998, Information technology – Security techniques – Hash-functions – Part 4: Hash-functions using modular arithmetic
16. Kargl, A., Meyer, B., Wetzel, S.: On the Performance of Provably Secure Hashing with Elliptic Curves. International Journal of Computer Science and Network Security 7(10), 1–7 (2007)
17. Lyubashevsky, V., Micciancio, D., Peikert, C., Rosen, A.: SWIFFT: A Modest Proposal for FFT Hashing. In: Nyberg, K. (ed.) Fast Software Encryption 2008, Proceedings. LNCS, Springer (to appear, 2008)
18. Menezes, A.J., van Oorschot, P.C., Vanstone, S.A.: Handbook of Applied Cryptography. CRC Press, Boca Raton (1997)
19. Merkle, R.C.: One Way Hash Functions and DES. In: Brassard, G. (ed.) CRYPTO 1989. LNCS, vol. 435, pp. 428–446. Springer, Heidelberg (1990)
20. Montgomery, P.L.: Modular Multiplication Without Trial Division. Mathematics of Computation 44(170), 519–521 (1985)
21. National Institute of Standards and Technology. FIPS PUB 180-1, Secure Hash Standard, April 17 (1995)
22. National Institute of Standards and Technology. FIPS PUB 180-2, Secure Hash Standard, August 1 (2002)
23. National Institute of Standards and Technology. Special Publication 800-57. Recommendation for Key Management – Part 1: General (revised) (March 2007)
24. Rivest, R.L.: The MD5 Message-Digest Algorithm, RFC 1321 (April 1992)
25. Rivest, R.L., Shamir, A., Adleman, L.M.: A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. Communications of the ACM 21(2), 120–126 (1978)
26. RSA Laboratories. PKCS #1: RSA Cryptography Standard (Version 2.1, June 14, 2002), http://www.rsa.com/rsalabs/node.asp?id=2125 [2008/1/3].
27. Wang, X., Lai, X., Feng, D., Chen, H., Yu, X.: Cryptanalysis of the Hash Functions MD4 and RIPEMD. In: Cramer, R.J.F. (ed.) EUROCRYPT 2005. LNCS, vol. 3494, pp. 1–18. Springer, Heidelberg (2005)
28. Wang, X., Yin, Y.L., Yu, H.: Finding Collisions in the Full SHA-1. In: Shoup, V. (ed.) CRYPTO 2005. LNCS, vol. 3621, pp. 17–36. Springer, Heidelberg (2005)
29. Wang, X., Yu, H.: How to Break MD5 and Other Hash Functions. In: Cramer, R.J.F. (ed.) EUROCRYPT 2005. LNCS, vol. 3494, pp. 19–35. Springer, Heidelberg (2005)