# A GOOD HASH FUNCTION IS HARD TO FIND, AND VICE VERSA

JOSHUA HOLDEN

Abstract. Secure hash functions are the unsung heroes of modern cryptography. Introductory courses in cryptography often leave them out — since they don't have a secret key, it is difficult to use hash functions by themselves for cryptography. In addition, most theoretical discussions of cryptographic systems can get by without mentioning them. However, for secure practical implementations of public-key ciphers, digital signatures, and many other systems they are indispensable. In this paper I will discuss the requirements for a secure hash function and relate my attempts to come up with a "toy" system which both reasonably secure and also suitable for students to work with by hand in a classroom setting.

## 1. Hash Functions

Hash functions are an essential part of modern cryptographic practice. At their root, however, hash functions don't necessarily have anything to do with cryptography, or even secrecy. By definition, a *hash function* is any function which takes an arbitrarily long string of characters or bits as input and returns a fixed-length output. A simple example using characters from the Roman alphabet is given in [2, Example 3.6.1, p. 233]: write the string in rows of five letters each (padding if necessary) and convert each letter to its numerical equivalent. Then add down the columns modulo 26 and convert the result back to letters. For example, if the input string is "Hello, my name is Alice", the procedure would go:

$$\begin{array}{ccccccccccc}
\mathrm{H} & \mathrm{E} & \mathrm{L} & \mathrm{L} & \mathrm{O} & \rightarrow & 07 & 04 & 11 & 11 & 14 \\
\mathrm{M} & \mathrm{Y} & \mathrm{N} & \mathrm{A} & \mathrm{M} & \rightarrow & 12 & 24 & 13 & 00 & 12 \\
\mathrm{E} & \mathrm{I} & \mathrm{S} & \mathrm{A} & \mathrm{L} & \rightarrow & 04 & 08 & 18 & 00 & 11 \\
\mathrm{I} & \mathrm{C} & \mathrm{E} & \mathrm{X} & \mathrm{X} & \rightarrow & 08 & 02 & 04 & 23 & 23 \\
\cline{7-11}
 & & & & & & 05 & 12 & 20 & 08 & 08 \\
 & & & & & & \downarrow & \downarrow & \downarrow & \downarrow & \downarrow \\
 & & & & & & \mathrm{F} & \mathrm{M} & \mathrm{U} & \mathrm{I} & \mathrm{I}
\end{array}$$

The output of the hash function, or *hash value*, is always five letters long, no matter how long the input is. In this case, "FMUII" would be the output.

When students first see hash functions, there are two common misconceptions that they often start with. First, hash functions are not encodings of the original string, in the sense that it is not possible to recover the string from the hash function, no matter how long you work at it or how clever you are. Since hash value has a fixed length while the input can be arbitrarily long, there simply isn't enough information to recover the original input. The second point that needs to be made is that hash functions are not in any way secret. There is no key to a hash function and anybody can compute the hash value given the input string.

Hash functions are used for purposes other than cryptography. In database design, for example, hash functions are used to convert a search key into a convenient index into a table where the record with that key is stored. A *checksum* is a hash function used to detect accidental errors introduced into a message or stored record. The hash function shown above makes a good checksum, since any single error will change the result, and two errors that canceled each other out would have to occur a multiple of five characters apart in the original message. It is rather unlikely that such a error would occur by accident!

There are two properties that are convenient for hash functions in most applications. First, hash functions should be fast to compute. Secondly, it is often useful for hash values to be distributed uniformly, it the sense that every possible hash value is used about the same number of times, even when the original set of inputs contains groups of similar strings. For example, if we are using a hash function as a checksum, we wouldn't want there to be many strings which all hashed to "AAAAA" and only a few that hashed to "ZZZZZ" — that would increase the chance that two similar strings would both hash to "AAAAA" and lessen the likelihood that we could detect errors between them.

For cryptographic purposes, however, these properties are not sufficient. For example, one common use of a hash function in cryptography is as part of a public-key digital signature. Suppose, as is usual in cryptography, Alice wants to send a message to Bob, in such a way that Bob knows that only Alice could have sent the message. Using a digital signature algorithm, Alice could cryptographically "sign" the message using a private key that only she knows. Bob could then use Alice's public key to verify that the message was properly signed. If Frank (a forger) wants to send Bob a message claiming to be from Alice, he would not be able to produce a properly signed message without the knowledge of Alice's private key.

However, secure digital signature algorithms are slow, and a larger input makes them slower. So instead of signing the entire message, Alice would like to sign a "message digest" which somehow captures the essence of the message but is much shorter. The output of a hash function is a good candidate for such a message digest, but only if the hash function is cryptographically secure.

Another use for a hash function is to get a message into the public record without revealing what it is. The use of anagrams for this purpose was common among scientists in the early modern period, when they wanted to make sure they got the priority for some discovery while still giving themselves time to refine it before publishing [7]. For example, after discovering what he thought were two large moons of Saturn, Galileo sent a letter to Kepler in 1610 with the sequence of letters:

smaismrmilmepoetalevmibunenugttaviras

Only after he was ready to publish his findings did Galileo reveal that these letters were an anagram for the Latin phrase *altissimum planetam tergeminum observari* (I have seen the uppermost planet triple) [7].

An anagram is not a true hash, since it does not have a fixed length output. A "lazy" anagram, however, simply alphabetizes the letters, which is equivalent to listing the number of times each letter in the message appears. Christiaan Huygens, for example, upon realizing that Galileo's "moons" were actually rings, in 1656 published the "anagram":

aaaaaaacccccdeeeeeghiiiiiiillllmmnnnnnnnnnooooppqrrstttttuuuuu

which contains exactly the letters in *annulo cingitur, tenui, plano, nusquam cohaerente, ad eclipticam inclinato* (it is surrounded by a slender flat ring, everywhere distant from the surface, and inclined to the ecliptic) [16, 17]. If, instead, Huygens had abbreviated his "anagram" something like the following:

a7 b0 c5 d1 e5 f0 g1 h1 i7 k0 l4 m2 n9 o4 p2 q1 r2 s1 t5 u5 x0 y0 z0

(leaving out, of course, the letters which do not appear in Latin) then he would have had a fixed length (if not particularly short) output and a true hash function. A hash function used for this purpose should again be cryptographically secure in order to prevent the discoverer from changing the meaning of his claim, or worse, publishing the hash value without any particular meaning in mind and making up the meaning later.

A *cryptographically secure hash function* should have three more properties besides the two mentioned above. First of all, it should be *one-way*, also known as *pre-image resistant.* Like other "one-way" functions in cryptography, this means that it should be computationally infeasible to compute an input to the hash function that would produce a given output.

(Note that I said "an input", rather than "the input"! Since the input could be arbitrarily long and the output length is fixed, there will be many inputs — infinitely many in theory — that produce the same output. These are called *collisions*.) If Frank gets a hold of a signed message digest for one of Alice's messages, he shouldn't be able to come up with a new message that has the same digest. If he did, then he would also have a signature for the new message, since the signature only depends on the digest. Bob wouldn't be able to tell that Alice had not signed the new message.

Furthermore, Frank shouldn't be able to come up with his new message even if he has Alice's original message. Hash functions that prevent this are called *second-preimage resistant*. In other words, if Frank has both the output of the hash function and an input that produces that output, it should be computationally infeasible for him to come up with a second input that also produces that output.

For many purposes, we would also like hash functions to be *collision-resistant*, meaning that it should be computationally infeasible for someone to find two inputs that give the same hash value. (Remember that collisions always exist for a hash function! But they should be difficult to find.) One application of collision-resistance is to prevent Bob from pulling a bait-and-switch on Alice. Suppose Alice agrees to buy Bob's car for \$5000. Bob draws up a contract and presents it to Alice for her signature. However, it turns out that Bob has found another message with the same digest. When Bob comes to collect his money, he presents the second contract, which says that she promised to pay him \$10000! Since both messages have the same valid signature, Alice can't prove that she really signed the one with the lower amount.

## 2. How Hash Functions Work

Cryptographers have come up with many different basic plans for secure hash functions. In fact, in the competition to choose a new standard hash function for the U.S. Government, the National Institute of Standards and Technology (NIST) specifically chose five finalists with different designs, so that an attack on one would be unlikely to disqualify the others as well [15]. For example, the five finalists included a "HAIFA" construction, two "wide-pipe Merkle-Damgård" algorithms, a "sponge" construction, and a "Unique Block Iteration" construction.

In fact, however, most of these constructions are conceptually descended from the *Merkle-Damgård* construction, which was invented in 1979 by Ralph Merkle. As shown in Figure 1, a Merkle-Damgård hash starts by splitting the message up into blocks (labeled M in the figure) of a fixed size. The message is then padded at the end ("length padding", labeled LP), not only to make sure there are an even number of blocks but also to incorporate the length of the message into the message itself. (This helps protect against a type of attack called a "length extension" attack.)

The next necessary ingredient is an *initialization vector*, or IV. The IV and the first message block are given as inputs to a *one-way compression function*, labeled as f in the figure. This compression function should have all of the same properties that we want the hash to have, except that instead of taking an arbitrary length input, it takes two fixed-length inputs, one of which is the same size as the fixed-length output. Thus the function "compresses" a fixed-length total input to a smaller fixed-length output. Ralph Merkle and Ivan Damgård each independently proved in 1989 that if the compression function in their

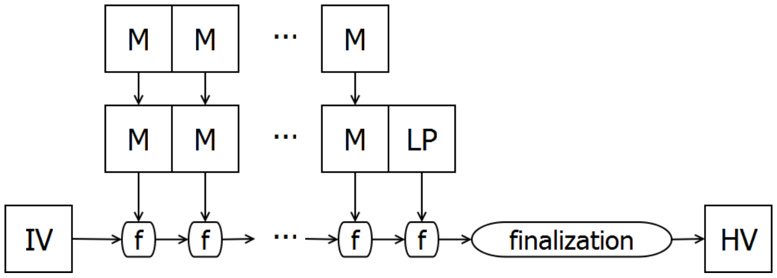

FIGURE 1. The Merkle-Damgård construction.

construction is collision-resistant, then the hash function is collision-resistant. The output of the compression function, called the *chaining value*, is used as one of the inputs to the next compression function. The next message block is the other input to the compression function, and this continues until all of the message blocks (including the padding) have been used. The final step in the construction is an optional “finalization”, which might truncate or further compress the last chaining value or might just add extra mixing. The output of the finalization is the hash value of the message (labeled HV).

Many hash functions use the Merkle-Damgård construction, with varying compression functions and numbers of bits used at each step. Some of the most popular include MD4 (Message Digest algorithm 4), invented by Ron Rivest in 1990, MD5 (Message Digest algorithm 5, an improved version of MD4), invented by Rivest in 1992, SHA (Secure Hash Algorithm), developed by NIST and the NSA and adopted as a U.S. government standard in 1993, and its successors SHA-1 (a slight tweak of SHA, 1995), and SHA-2 (a major revision, 2001).

We will take MD5 as a representative example. (The official specification of MD5 can be found in [9].) Each message block is 512 bits long, while the initialization vector, which is specified by the official description, is 128 bits long. The compression function takes a 512-bit input and a 128-bit input and produces a 128-bit output. The overview of the compression function is shown in Figure 2. Inside the compression function the chaining value is split into four 32-bit words, labeled A, B, C, and D in Figures 2 and 3, and the message block is split into 16 32-bit words. The function is split into four “rounds”, each of which is composed of 16 “steps”. One step is shown in Figure 3.

In each step, words B, C, and D of the chaining value are combined using a non-linear bitwise function (labeled F in Figure 3) which changes depending on the round. The result of this is added to word A of the chaining value, a word of the message block (labeled M) and a 32-bit constant (labeled K) which is different for each step. (Each word of the message block is used once in each round, with the orders varying.) All these additions are done modulo $2^{32}$. The bits of this result are then rotated by a number of places which depends on the step and added modulo $2^{32}$ to the word B of the chaining value. Finally, this result becomes word B of the chaining value and each other word rotates one place to the right. After the last step of the last round, the initial words of the chaining value as they came into the compression function are “fed forward” and added modulo $2^{32}$ to the new values to produce the final chaining value. There is no special finalization step after the last message block is processed.

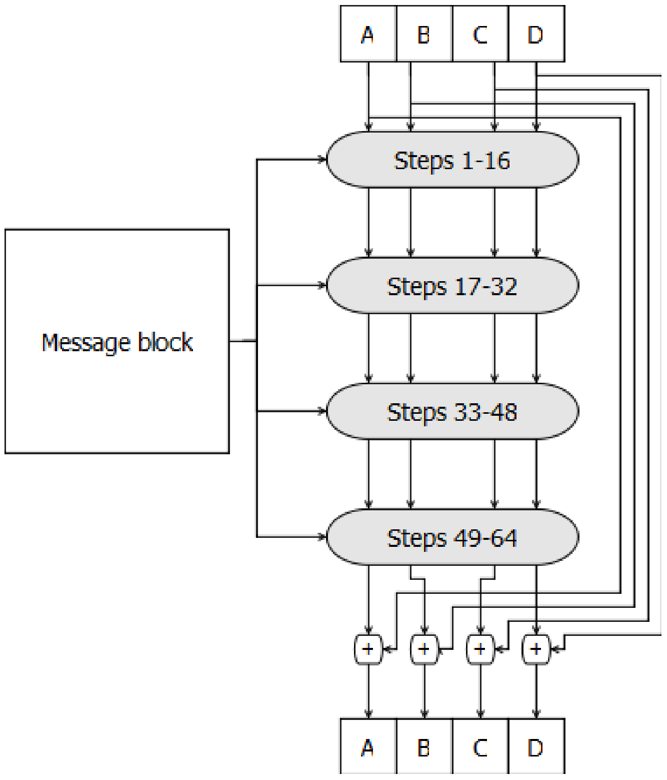

FIGURE 2. Overview of the MD5 compression function.

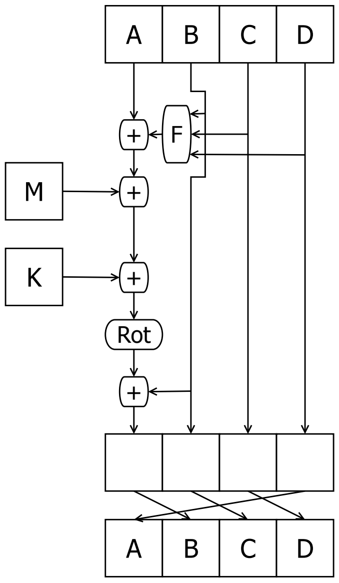

FIGURE 3. One step of the MD5 compression function.

Key ingredients to note in MD5 are the combination of linear and nonlinear functions and the combination of functions which act on bits and functions which act on words. Also note that words are frequently combined to provide diffusion, and that there is feed-forward from the beginning to the end of the compression function.

## 3. Teaching about Secure Hash Functions

During the Fall of 2000, I taught a course on "Cryptography and Society" at Duke University. (See [4] for more details of this course.) I had a section on RSA digital signatures, and since one of themes of the course was how cryptography was used in practice, I wanted to talk about secure hash functions. In addition, the course was designed for students not majoring in math or computer science, so I wanted a hash function which would be easy for such students. Since there were not a lot of resources for such courses at the time, I simply made up my own hash function, which I later called "Josh's Hash Algorithm", or JHA.

My goals for this function were:

**Simplicity:** Students with limited experience should be able to do exercises with it without a computer in a single class period, and

**Security:** It should be reasonably secure (preimage-resistant, second-preimage resistant, and collision-resistant) given the previous goal.

JHA takes a string of letters and spaces and outputs an integer between 0 and 17 according to the following rule:

$$\text{hash} = (7 \times \#\text{ of vowels} - 3 \times \#\text{ of consonants} + \#\text{ of spaces}^2) \bmod 17$$

For example, the string "Hello my name is Alice" has a hash value of

$$(7 \times 8 - 3 \times 10 + 4^2) \bmod 17 = 8.$$

How did this hash stack up against my design goals? Well, it did about as well with regard to simplicity as one could hope for, although some decision had to be made with regard to the letter "y", and some students had to be reminded that the number of spaces was not the same as the number of words. (These issues actually may have been good reinforcements of the point that the specification of a hash function has to be public and unambiguous!) JHA was less good with regard to security, however. It's moderately secure with respect to preimage attacks, especially if you restrict your message space to sensible English text. It is not very secure with respect to second-preimage and collision attacks, however. It is often possible to change one or a few vowels to other vowels and consonants to other consonants in an English text and still get a sensible text, perhaps with a very different meaning!

In January 2010 I taught a similar course on cryptography and society at Manchester College, and in May 2010 I taught a general education course on cryptography at Goshen College, both in northern Indiana. Once again I wished to have a hash function with the two goals above. By this time Barr's textbook ([2]) had come out, and I was using it for my main cryptography textbook in both courses. I referred above to one of the hash functions described in that book, which also does fairly well with regard to the goal of simplicity.

However, as Barr points out in his Exercises ([2, Exercise 3.6.3, p. 241]), the function is not particularly one-way, especially if one does not restrict the message space. The problem is that there is no diffusion between the five columns, and that the operation is completely linear.

Barr offers another relatively simple hash function in [2, Example 3.6.2, p. 236], this one using essentially a Merkle-Damgård structure on 8-bit blocks, but without length padding. The compression function is a bitwise exclusive-or of the chaining value with the message block followed by reversing the bits of the block in adjacent pairs. This is considerably more secure than either of the hashes above, but I rejected it as too complicated for my audience and classroom goals.

In the end, I went with a slightly modified version of JHA, which I called JHA-1:

$$\text{hash} = 5^{(7\times\#\text{ of vowels}-3\times\#\text{ of consonants}+\#\text{ of spaces}^2)} \bmod 17$$

For example, the string "Hello my name is Alice" has a hash value of

$$5^{(7\times 8-3\times 10+4^2)} \bmod 17 = 9.$$

JHA-1 is even more secure with respect to preimage attacks, since they would involve solving a (small) discrete logarithm problem, but it's really no better with regard to the other types of attacks. However, it admirably fit my simplicity goal. While teaching at Goshen College, JHA-1 allowed me to do a classroom exercise where students wrote a short message, computed its hash value, and then signed it with their own individual (small) RSA key. (Students used a single computer present in the classroom to do the modular exponentiation steps in JHA-1 and RSA.) As the students finished, they were instructed to put a slip of paper with the message and signature in a pile and draw out a similar slip from another student. Using the RSA public keys, which were listed on a blackboard in the classroom, the students verified each others' signatures. Along the way students sometimes discovered mistakes which might have occurred in any of the hashing, signing, and verification steps, but overall I felt the exercise was a success.[1]

## 4. JHA-2

With these experiences in mind, I decided in the Spring of 2011 to see if I could design a better hash function for use in the classroom. In particular, I wanted to see if I could make a function that was still fairly simple but displayed some resistance to second-preimage and collision attacks. After some consideration I also added the two following design goals:

[1]This course met four days a week with most days consisting of a one-hour lecture, a one-hour break, and then a two-hour block which was usually divided into about an hour of lecture and an hour of hands-on work. One entire day was devoted to digital signatures and hash functions, with a general introduction and the idea of RSA signatures before the break. After the break I covered the hash function from [2, Example 3.6.1, p. 233] mentioned above, JHA-1, examples of hash functions and digital signatures, and the classroom exercise. The students were already familiar with RSA encryption at this point.

**Realism:** It should use elements from hash functions that are actually used in the "real world", and

**Optimization:** It should be "optimized" to be easy to compute with the aid of a four function calculator.[2]

While doing background research for this project, I came across three more "toy" hash functions designed for classroom use, in addition to the ones from Barr's book. (I'm sure there are more out there!) Trappe and Washington ([14, Section 8.2, pp. 222–223]), and Stallings ([11, pp. 336–337]) both give a hash function operating on $n$-bit blocks. This is similar to Barr's first hash except that addition is done modulo 2 and each row is rotated one bit to the left before the addition. Both authors, however, acknowledge that this is still very insecure owing to the lack of diffusion and nonlinearity.

Stallings ([11, Exercise 12.4, pp. 375–376]) also gives a description of the "Toy Tetragraph Hash", which he describes as "similar in spirit" to the Merkle-Damgård hashes I listed above. It uses 16-letter message blocks and chaining values consisting of four numbers modulo 26, and thus also has some similarities to Barr's first hash. It uses two rounds in the compression function, however, and the second round adds diffusion by permuting each row in a different manner before the message block is added once again. This was a very intriguing possibility, and it did satisfy the goal of realism, but I eventually rejected it on two grounds. First, the lack of nonlinearity leads to some weakness with respect to preimage attacks, as pointed out by Stallings. And second, the many modulo 26 computations seemed to me to be two difficult to perform on a four-function calculator, as required by the goal of optimization.

Stamp ([12, Exercise 5.9.16, p. 106]) mentions a "Bobcat" hash algorithm that he created as a version of the 192-bit Tiger hash [1] scaled down to 48 bits. This does not seem suitable for use without a computer. Stamp also mentions a 12-bit version of the hash. It appears that this is merely the 48-bit hash with the output truncated to 12 bits as part of the finalization step. It might be interesting to develop an actual 12-bit version of Tiger/Bobcat, along the lines of the Simplified DES [10], Simplified AES [8], and Demitasse (a simplified TEA) [5] algorithms.[3] However, based on my experiences in the courses described in [4] and [5], I think that this might be pushing the limits of what my target audience could adequately carry out, in addition to violating the optimization design goal.

So I went back to the proverbial drawing board. The goal of realism suggested a Merkle-Damgård design, while optimization for four-function calculators suggested using decimal digits rather than bits, and moduli that were powers of 10 rather than 26. The goal of simplicity led me to use chaining values that were only two (decimal) digits long, and to limit the number of rounds. The goal of security led to the use of both linear and nonlinear functions, and to mix functions that operated on digits with those that operated on two-digit numbers modulo 100. The result was a hash function I call JHA-2.

Each message block of JHA-2 consists of one letter, which is converted into two decimal digits in the usual way. Length padding is added to the end in the form of a two-digit number

[2]I found in my classes in 2010 that despite more powerful calculators being fairly cheap and convenient, nontechnical students often did not have them or did not bring them to class. However, most of them had a cell phone, and most cell phones now include a four-function calculator!

[3]I suggest the name "Housecat" for the 12-bit version.

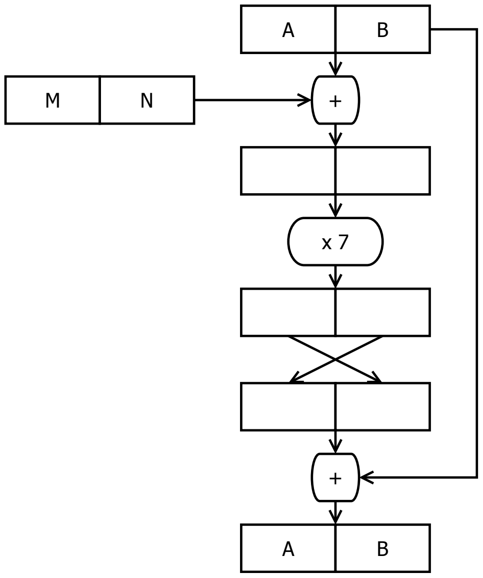

FIGURE 4. The JHA-2 compression function.

indicating the number of letters in the message. So if the input string is "Hello, my name is Alice", the message would become:

| | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| H | e | l | l | o | m | y | n | a | m | e | i | s | A | l | i | c | e | |
| 07 | 04 | 11 | 11 | 14 | 12 | 24 | 13 | 00 | 12 | 04 | 08 | 18 | 00 | 11 | 08 | 02 | 04 | 18 |

I chose the initialization vector for JHA-2 to be 76.[4] There is no special finalization for the hash. The compression function of JHA-2 consists of a single round, which proceeds as follows (see Figure 4):

(1) Add the chaining value (labeled AB in Figure 4) to the new message block (labeled MN) modulo 100.
(2) Multiply the chaining value by 7, modulo 100.
(3) Reverse the digits of the chaining value.
(4) Add the initial digits of the chaining value to the new chaining value modulo 100.

Note that the first, second, and fourth steps are linear regarded modulo 100, but are not linear in the tens digit modulo 10. Conversely, the third step is linear in each digit modulo 10, but is not linear modulo 100. The second step adds extra diffusion.[5] The first step is of course necessary to mix in the message block and the fourth step is feedforward.

Continuing with our earlier example, the beginning of the calculations would proceed:

[4]For 1976, the year in which the famous paper "New Directions in Cryptography" [3] was published.

[5]An early version without this step was shown to be insecure by Michael Pridal-LoPiccolo, an undergraduate at Rose-Hulman who was able to find two-letter preimages.

$$\begin{array}{rcc l}
 & 7 & 6 & \\
+ & 0 & 7 & \text{(new block)} \\ \cline{2-3}
 & 8 & 3 & \\
\times & & 7 & \\ \cline{2-3}
 & 8 & 1 & \\
 & \multicolumn{2}{c}{\longleftrightarrow} & \\
 & 1 & 8 & \\
+ & 7 & 6 & \text{(feedforward)} \\ \cline{2-3}
 & 9 & 4 & \\
+ & 0 & 4 & \text{(new block)} \\ \cline{2-3}
 & & \vdots &
\end{array}$$

And the results after each message block would be:

| | H | e | l | l | o | m | y | n | a | m | e | i | s | A | l | i | c | e | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 07 | 04 | 11 | 11 | 14 | 12 | 24 | 13 | 00 | 12 | 04 | 08 | 18 | 00 | 11 | 08 | 02 | 04 | 18 |
| 76 | 94 | 62 | 73 | 61 | 13 | 70 | 55 | 22 | 67 | 02 | 26 | 09 | 07 | 01 | 49 | 48 | 53 | 52 | **61** |

The final value of the hash is therefore 61.

## 5. Evaluation of JHA-2

How well does JHA-2 fit its design goals? I have not yet had a chance to teach a class using it, so I can't say for sure whether it is simple enough to use in an in-class exercise, but based on previous experience I have high hopes. (If any readers try it out for themselves, please let me know how it went!) I timed myself on the 11-letter string "This is a test" and it took me about 6 minutes with the aid of a table of letter-to-number equivalences and a four-function calculator. Unfortunately, it took me three tries to get it right! By the third try I had gotten it down to 4 minutes, and established that the most common mistake for me was adding the wrong number for the feedforward step. I tried various work grids in order to combat this, and the one that seemed to work best looked like:

| | T | h | i | s | i | s | a | t | e | s | t | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| feedforward | 76 | 32 | $\cdots$ | | | | | | | | | 45 |
| message | 19 | 07 | 08 | 18 | 08 | 18 | 00 | 19 | 04 | 18 | 19 | 11 |
| sum | 95 | $\vdots$ | $\ddots$ | | | | | | | | | 56 |
| times 7 | 65 | | | | | | | | | | | 92 |
| reverse | 56 | | | | | | | | | | | 29 |
| plus feedforward | 32 | | | | | | | | | | 45 | **74** |

The experience was also very instructive in terms of illustrating an important feature of hash functions of this sort — a single change quickly propagates through the calculations! This is desirable, because it leads to the *avalanche effect*, namely that a small change in your input results in a large change in the output. This makes it more difficult to mount the sorts of attacks we have discussed. However, it also means that once you make your first mistake, generally none of the rest of your work can be salvaged.

My calculations of hash values also suggested that the algorithm was doing well with regard to optimization for four-function calculators. It should be pointed out to students reducing the chaining value modulo 100 is the same as dropping all but the last two digits, and that it does not matter whether this is done for every operation or only selected ones. If the student takes this to heart, and the calculator operates in the common fashion of ignoring arithmetic precedence and storing each value in an accumulator as it is calculated, then the only point at which the accumulator (i.e., chaining value) needs to be reinitialized is when the digits are reversed.

As far as realism is concerned, I mentioned earlier that JHA-2 uses the Merkle-Damgård construction, linear and nonlinear functions, diffusion, and feedforward. A constant could be added to the chaining value during each round, like in MD5, but I decided it would add too much complication for too little value in security and realism.[6]

Security of hash functions, like that of ciphers, is generally measured by the time necessary to carry out an attack by an opponent with a specified set of resources. Since JHA-2 was designed to be feasible without a computer, we should probably not expect it to stand up to attacks using a computer. In fact, since there are only 100 possible hash values, a brute force attack would be expected to find a preimage or second preimage after 100 tries on average. This is easily done with a computer but probably beyond the patience of even a dedicated student. On the other hand, an entire class of 20 to 30 students might reasonably attempt it using short inputs. This might illustrate that JHA-2 could be considered secure against individuals but not against medium-sized groups.

For a collision attack, however, we are interested in finding *any* two messages with the same hash value, with nothing specified. This is much easier, and in fact the "birthday paradox" effect says that we would expect a collision to occur after an average of approximately $\sqrt{(\pi/2)H}$ tries, where $H$ is the number of hash values. With 100 hash values, this comes out to approximately 12.5 tries on average, which is just within the realm of possibility for a single individual — perhaps as a homework problem for a particularly dedicated student! This probably still makes JHA-2 secure enough to deter the casual opponent, however. If more security against brute-force attacks was desired, it would not be hard to extend the hash function to three, four, or even more digits in the hash value and chaining value. In such a case, the multiplication by 7 should probably be changed to a larger value. (I would suggest no fewer digits than one less than in the chaining value, for maximum diffusion.) Also a decision would have to be made on how to permute the digits in the next-to-last operation of the round. (A rotation of one place to the right would mirror the choice made in MD-5.)

JHA-2 is also (of course) vulnerable to other attacks which apply generically to all hash functions based on the Merkle-Damgård construction, such as the "Nostradamus attack". (See [6] or [13, Section 5.2.4].) This attack relies on collisions in the underlying compression function to find "chosen target forced prefix" preimages — i.e., given a target hash value

[6]On the other hand, the Tiger hash function [1] uses multiplication by a different constant in each round, rather than addition. So perhaps the multiplication by 7 in JHA-2 could be looked on in this manner. It should also be noted, however, that much of the security purpose of the constant comes from not using different constants in different steps or rounds, in order to eliminate symmetry that could be exploited.

the attack can find a preimage message which starts with a specified prefix and ends with a random but (more or less) meaningful suffix. It is likely that differential attacks such as those which have been discovered against MD4 and MD5 (see [13, Sections 5.3 and 5.4]) and more recently SHA and SHA-1 are also possible against JHA-2, although I have not worked them out in detail. Also, I doubt that they would be faster than brute force against the unmodified algorithm, although they could very well be practical against a version with more digits in the hash value. Working out the details of such an attack seems like an excellent student project and I look forward to hearing of such a thing!

## 6. Conclusion

The goal of this project was certainly not to develop a state-of-the-art secure hash function. Nor was it necessarily to develop a simplified version of a common hash function, similar to what was done for block ciphers in [10], [8], and [5], although that might also be an interesting project. Rather, the goal was to create a hash function that was similar enough to common hash functions to be realistic while still being practical for students with limited mathematical and computer backgrounds. The proof of such a thing can only come from classroom experience, of course. Hopefully one of you reading this will soon be in a position to give it such a test. I eagerly await the result!

## References

1. Anderson, R. and Biham, E. 1996. *Tiger: A Fast New Hash Function.* Fast Software Encryption: Third International Workshop (Cambridge, UK, February 21–23, 1996), Proceedings (Gollmann, D., ed.), LNCS, vol. 1039, Springer, Berlin, pp. 89–97.
2. Barr, T. H. 2001. *Invitation to Cryptology*, Upper Saddle River, NJ: Prentice Hall.
3. Diffie, W. and Hellman, M. E. 1976. *New Directions in Cryptography.* IEEE Transactions on Information Theory. 22(6): 644–654.
4. Holden, J. 2004. *A Comparison of Cryptography Courses.* Cryptologia. 28(2): 97–111, DOI 10.1080/0161-110491892809.
5. ———. 2013. *Demitasse: A "Small" Version of the Tiny Encryption Algorithm and its Use in a Classroom Setting.* Cryptologia. 37(1): 74–83, DOI 10.1080/01611194.2012.660237.
6. Kelsey, J. and Kohno, T. 2006. *Herding Hash Functions and the Nostradamus Attack*, Advances in Cryptology — EUROCRYPT 2006 (Vaudenay, S., ed.), LNCS, vol. 4004, Springer, Berlin, pp. 183–200.
7. Meadows, J. 2007. *Limitations on the publishing of scientific research*, The History of Information Security (Leeuw, K. De and Bergstra, J., eds.), Elsevier Science B.V., Amsterdam, pp. 29–51.
8. Musa, M. A., Schaefer, E. F., and Wedig, S. 2003. *A Simplified AES Algorithm and its Linear and Differential Cryptanalyses.* Cryptologia. 27(2): 148–177, DOI 10.1080/0161-110391891838.
9. Rivest, R. April 1992. *The MD5 Message-Digest Algorithm*, Technical Report 1321, Request for Comments, Internet Engineering Task Force, available at `http://tools.ietf.org/html/rfc1321`.
10. Schaefer, E. F. 1996. *A Simplified Data Encryption Standard Algorithm.* Cryptologia. 20(1): 77–84, DOI 10.1080/0161-119691884799.
11. Stallings, W. 2005. *Cryptography and Network Security*, 4th ed., Upper Saddle River, NJ: Prentice Hall.
12. Stamp, M. 2005. *Information Security: Principles and Practice*, Hoboken, NJ: Wiley-Interscience.
13. Stamp, M. and Low, R. M. 2007. *Applied Cryptanalysis: Breaking Ciphers in the Real World*, Hoboken, NJ: Wiley-IEEE.
14. Trappe, W. and Washington, L. C. 2005. *Introduction to Cryptography with Coding Theory*, 2nd ed., Upper Saddle River, NJ: Prentice Hall.

15. Turan, M. S., Perlner, R., Bassham, L. E., Burr, W., Chang, D., Chang, S., Dworkin, M. J., Kelsey, J. M., Paul, S., and Peralta, R. February 2011. *Status Report on the Second Round of the SHA-3 Cryptographic Hash Algorithm Competition*, Technical Report 7764, NIST Interagency Report, NIST, available at `http://csrc.nist.gov/groups/ST/hash/sha-3/Round2/documents/Round2_Report_NISTIR_7764.pdf`.
16. van Helden, A. 1974. *"Annulo Cingitur": The Solution to the Problem of Saturn.* Journal for the History of Astronomy. 5: 155–174.
17. van Maanen, A. 1926. *Review: Ouvres Compltes de Christiaan Huygens, Vol. XV.* The Astrophysical Journal. 63: 375–376.

## Biographical Sketch

Joshua Holden is currently an Associate Professor in the Mathematics Department of Rose-Hulman Institute of Technology, an undergraduate engineering college in Indiana. He received his Ph.D. from Brown University in 1998 and held postdoctoral positions at the University of Massachusetts at Amherst and Duke University. His research interests are in computational and algebraic number theory, cryptography, and the application of graph theory to fiber arts. His teaching interests include the use of technology in teaching and the teaching of mathematics to computer science majors, as well as the use of historically informed pedagogy. His non-mathematical interests used to include fiber arts, but that now seems to be a mathematical interest. Still largely in the non-mathematical category are his interests in science fiction and music, both classical and contemporary.

Department of Mathematics, Rose-Hulman Institute of Technology, Terre Haute, IN 47803, USA
*E-mail address*: holden@rose-hulman.edu