SHA

Timeline for SHA

1993 NSA publishes Federal Information Processing Standard (FIPS) 180: SHA(0).[1]
- Intended for use with DSS.

1995 FIPS 180 is withdrawn without comment and FIPS 180-1 is published (SHA-1).[1]

2002 FIPS 180-2 is accepted by NIST (to begin implementation from FEB 03).[2]

2004 November: 2nd preimage resistance broken. Complexity of break 2^{106} as opposed to 2^{160} for brute force.[3]

2005 February: Collision Resistance broken - Complexity of break less than 2^{69} (as opposed to 2^{80} for brute force).[4]

2005 August: Break optimized - now complexity is 2^{63}. [5]

2010 Deadline for the switch from SHA-1 to SHA-2 set by NIST[2]

Known Implementations
SHA is used in many different places, governmental as well as private.

A few samples [6]
- SSL,
- PGP,
- TLS,
- SSH,
- S/MIME,
- IPSec,
- and of cause DSS.

Why use hashing
With DSS, SHA-1 was/is primarily used for hashing the message and then signing the hash instead of the message itself. The advantage of this is that you only have to perform the rather expensive cryptographic functions on a (in this case) 160 bit long string even if the message itself is much, much longer. This reduces the necessary bandwith and CPU-power at the servers that are needed to verify a signature.
This requires a hashing algorithm that's 2^{nd} preimage resistant and, if certain criteria can't be met, collision resistant, but a lot more about this later.

Cryptanalysis

Definitions

There are 3 important definitions we need for the analysis:

2nd preimage resistance:
Given a digest d and a message m, find a message h(m') = d
Commonly prevented through the ”avalanche” effect

Collision resistance:
Find two arbitrary messages m and m’, such that h(m) = h(m'). This is by far the easiest attack (not that it is easy though).

A break:
Any algorithm/theorem that finds collisions/proves that they can be found and has less complexity than brute force.

Analysis of SHA

First we need to get some idea as to the complexity of brute forcing SHA. By using the birthday “paradox”, it can easily be shown that the collision attack complexity of any digest is $2^{|digest|/2}$ which means 2^{80} for SHA-1 and 2^{112}, 2^{128}, 2^{192} and 2^{256} respectively for SHA-2.

Actually, it has been shown[8] to be $1.17 * 2^{|digest|/2}$ but since this is a small constant, I will ignore it.

What does this mean then? Perform 2^{80} operations and you can do anything? No, it means that in order to, with probability ½, find a single collision, you would need to perform this many operations. You could find a collision in the first try, and you may never find it (never being 2^{160} in this case).

On a personal note (I haven’t found anyone proving this), the space complexity of brute forcing should be the same. The reasoning is, that if you put 23 people into a room at the same time, you would get that nice ½ prob., but if you only have two persons in the room at any one time and exchange them when you have tested their birthday, you would change the probability. Hence it should be necessary, not only to remember each previous hash during the brute force, but also to compare each of these hashes with the current one. This can be done in constant time using infinite memory, but in the real world it is a problem.

Hence brute forcing SHA-1 should be more or less impossible today. Then again, I might have forgotten something.

If we instead look at 2nd preimage resistance for SHA-1 (and most other hash-algorithms) it has been proven[3] to be at least 2^{160} (the length of the digest) for brute force attacks.

Example time needed to hash messages (1 GHz Pentium III) [8]

null string	= 0.027 sec
”hi”	= 0.024 sec
”please hash me, I would really like to be hashed. HASH ME! Please?”	= 0.037 sec
1.000.000 a’s (digest present in FIPS 180-1)	= 0.072 sec*
114 MB file	= 7.830 sec

* - note that I tried to recalculate one value based on the others. I assume the one in the article is a type.

The Attacks

I am going to discuss two attacks, the first being a 2^{nd} preimage attack and the second a collision attack.

The first one[3] was presented by Bruce Schneier and John Kelsey in 2004. It wasn’t really an attack on SHA, but more an attack on any algorithm using the Damgard-Merkle structure. It proves that it is possible to find 2^{nd} preimages in less than 2^{106} hashes as opposed to the 2^{160} for brute force.

The second[4] is from Xiaoyun Wang, Yiqun Yin and Hongbo Yu who in 2005 described a way to break the collision resistance of SHA-1 (among others), successfully reducing the complexity from 2^{80} as to 2^{63}.

Before actually looking at the consequences, I would like to argue why it is even necessary to look at these attacks. The first one is pretty obvious. If 2^{nd} preimage is broken, it is possible to make someone sign a contract of any kind, and then change it to something else, by creating a second document that hashes to the exact same digest. This would kill DSS and basically any protocol that uses secure hashing.

Collision attacks, on the other hand, aren’t as powerful. In order to cheat here, you would have to create two different documents, both with the same hash, and then trick someone into signing one, in effect, signing the other as well.

It should be clear that it is a threat, so let’s look at the severity.

Why we shouldn’t start worrying

The first attack has one major flaw (with regard to practical use); it assumes that we have unlimited memory available, and even if this is possible in theory (through swapping and _many_ tapes or disks), it isn’t possible today. I found it difficult to extract any exact values, but it seems that between 2^{48} and 2^{80} blocks are necessary in order to achieve the above mentioned complexity.
Assuming that it was actually possible to create this much fast memory in a single system, we have another problem.

In 2002, a search for a single key[9] (for a cryptographic function, not for hashing) with

complexity 2^{64} ended in success after 1757 days. If this is used to give a general idea of how complex a problem we are looking at, then 2^{106} hashes is far beyond our reach today. At least for any single person, but in most likelihood, even for the NSA.

So it seems 2nd preimage attacks are out of our reach. How about just finding a collision then?

Well, the Chinese team proved that it was possible to find collisions in less than 2^{63} hashes (with prob. ½), which is well below brute force. Also, their algorithm uses a constant amount of space, so it is actually possible to run it. They have even found collisions in SHA-1 reduced to 58 rounds (with complexity 2^{33}).

If we assume the average PC can compute 1000 hashes pr. second, time needed for finding one collision would be (very, very roughly) 292 x 10^6 years, so it isn't really feasible for the single nerd with too much time on his/her hands.

This should be compared to 38 x 10^12 years for brute force, though.

To the best of my knowledge (and web-searching abilities), no collision has ever been documented within SHA-1, and it isn't likely that one will be in the near future, considering the complexity. It is, in practice, possible to create a collision in SHA-1 in around 5 years, if we use the above example as a guideline. This should be taken as a threat, so let's look at the damage.

The only thing you need to recreate security without collision resistance is to add something of you own to any document you sign. If you for instance add the time of signing and transmit this to the recipient together with the signature, they could add it to their own document and verify the signature. On the other hand, it is extremely unlikely that they would actually be able to guess this beforehand and then include it in the second document (thus upgrading the problem from collision resistance to 2nd preimage resistance).

So even if it was possible to break the collision resistance, security can easily be upgraded.

Also, with SHA-2, since the digest increases in size, so should the complexity of the attack. It isn't actually mentioned in the article, but it is reasonable to assume.

Why we should start worrying

We now have attacks that both reduce the collision resistance and the 2nd preimage resistance of SHA. With SHA-1 one of the attacks actually makes it possible to create collisions. Not meaningful collisions, but still.

It is reasonable to assume that people will build on these attacks. It is also reasonable to assume that some agencies (e.g. the NSA) and some corporations may already have known about these attacks for some time, and have already reached the next level.

It is now proven to be possible to reduce the complexity of two different attacks well below brute force for SHA. It hasn't been proven that they can be broken in the true sense of the word, but we are getting there and know we have pointers that show an attack may be possible someday.

NSA and other equivalent organisations may already be able to perform these.

Conclusions

Based on what I have presented here, in my opinion, SHA-1 is still secure enough to be used in most applications until 2010 as recommended in FIPS 180-2. Implementations should however be tested against collision attacks, and if prevention isn't possible, they should be upgraded to SHA-2 now.

It is possible that in the near future, SHA-1 and perhaps any algorithm based on the Damgard-Merkle construct will be broken for real, but in this case SHA-2 won't help. Therefore it is time that we begin designing new hash-algorithms based on new ideas.

But, as it was said in one of the articles: "Even broken, SHA-1 is still almost as secure as MD5 ever was".

References:

1: http://www.itl.nist.gov/fipspubs/fip180-1.htm
2: http://csrc.nist.gov/publications/fips/fips180-2/fips180-2withchangenotice.pdf
3: http://eprint.iacr.org/2004/304.pdf
4: http://evan.stasis.org/odds/sha1-crypto-auth-new-2-yao.pdf
5: http://www.k2crypt.com/sha1.html
6: http://en.wikipedia.org/wiki/SHA1
7: http://cm.bell-labs.com/who/akl/hash.pdf
8: "Secure Hashing Algorithm family of secure hashes" - Mike Adams
9: http://www.schneier.com/blog/archives/2005/02/cryptanalysis_o.html

Appendix A: The SHA-family [2, 8]
In most of the following, I will disregard SHA-0 since it was long ago deemed unfit for existence. I will, however, point out the differences between SHA-0 and SHA-1 when we get to the description of the algorithm.

The latest member of the SHA family is SHA-2. It consists of SHA-224 (not in the initial FIPS), SHA-256, SHA-384 and SHA-512.

Basically, these have the same structure, being iterative hash algorithms, but some of the SHA-2 algorithms only have an 63-round chaining block digest, where the others and SHA-1 use 80 rounds. Also, where SHA-1 uses 4 different constants(described later) for the 80 rounds, SHA-2 uses 63, respectively 80 instead. This is mainly to ensure more obscurity and to increase the avalanche effect on each hash.

The numbers assigned to each version of SHA-2 describes the length of the resulting hash. As I will exploit during the analysis, this length can be translated directly into the brute force security of the algorithm.

SHA-1 and the first two algorithms in SHA-2 all use blocks of length 512 bit. The last two use blocks of length 1024 instead. At the same time, these last two can hash messages of length $< 2^{128}$, where the first three are limited to 2^{64}.

How does it work[2]
SHA-1 works much like cooking a meal; at first we take whatever groceries we've bought and prepare them for cooking.
Then we take the most important parts, e.g. a roast, and prepare them, using the lesser important once to obscure how boring a roast really is and the make the meal seem like more than it really is.
Finally, we devour the meal in its entirety, changing it into something completely unrecognizable from the initial product.

SHA-1 does the same, in the following three fases:
- Initialization (run once)
- Expansion (run once per block)
- Digestion (run 80 times per block)

Initialization
First we need to pad the message and append information about the length. Example used is the message m = "x6162636465".

First we simply append a 1-bit. Then we fill with zeroes until the length of the message is (|blocks| * 512) – 64. Lastly we add information about the length (64 bits). This is done to prevent several different attacks, including the "Long Message Attack".

Then we initiate our "digest buffer", which consists of 5 32-bit words, with the values

"x67452301", "xEFCDAB89", "x98BADCFE", "x10325476" and "xC3D2E1F0". Where these actually come from (or more specifically, the reasoning behind them) I do not know. As far as I can tell, they were first presented in Ronald Rivest's MD5 algorithm. Also note that these have been changed for SHA-2.

A small personal observation is that if you want to obscure your hash-algorithm, you can change these values and then use them as keys to recreate a certain hash. This might be useful to prevent others from learning exactly which in a row of public documents you have signed, but I don't think it's used in the real world.

Expansion
The message is then split into its blocks and these are processed one by one. Each block is first bloated into 2560 bits (SHA-1, more for the rest) that are then digested one at a time.

First we create a table W with room for the 80 words. Then we enter the first 16 words (the block from the message) into the table. These are then used to fill out the rest of the table, using the function $W[t]=ROTL^{1}(W[t-3] \text{ xor } W[t-8] \text{ xor } W[t-14] \text{ xor } W[t-16])$.

Finally, the Chaining Values, A, B, C, D and E, are initiated with values corresponding to H0 through H4.

Digestion
Now we go through the 80 rounds in which we digest the bloated message. In order to achieve a proper avalanche-effect, we need more bits. SHA-1 uses "80" constants (one pr. round) and performs different bitwise operations in each round.

The 80 constants in SHA-1 are defined as follows K[t] (where t is an integer between 0 and 79):
"x5A827999" for $0 \le t \le 19$
"x6ED9EBA1" for $20 \le t \le 39$
"x8F1BBcDc" for $40 \le t \le 59$
"xCA62C1D6" for $60 \le t \le 79$

As mentioned before, here it is worth noting that in SHA-2, 63 or 80 different constants are actually used.

The function $F(t, B, C, D)$ (with t as above and B, C and D as 32-bit words) is defined as
$(B \wedge C) \vee ((\neg B) \wedge D)$ for $0 \le t \le 19$
$B \text{ xor } C \text{ xor } D$ for $20 \le t \le 39$
$(B \wedge C) \vee (B \wedge D) \vee (C \wedge D)$ for $40 \le t \le 59$
$B \text{ xor } C \text{ xor } D$ for $60 \le t \le 79$
This is the actual digest-function. It reduces 96 bits to 32.

As we go through each round, each chaining value is moved one place (e.g. B = A) after

$TMP = ROTL^5(A) + F(t, B, C, D) + E + W(t) + K(t)$ has been calculated.

This is then stored in A and we proceed to the next round. It is worth mentioning, though, that when we switched from SHA-0 to SHA-1, C = B was changed to C = $ROTL^{30}(B)$. Why exactly this was done, isn't publically available, but it is assumed it was done to remove some of the vulnerabilities later found in MD5.

After the digest, we add the chaining values to our digest buffer and continue with the rest of the message. When we are done, the digest buffer is concatenated into the digest itself.