Compare-by-Hash: A Reasoned Analysis

J. Black [*]

April 14, 2006

Abstract

Compare-by-hash is the now-common practice used by systems designers who assume that when the digest of a cryptographic hash function is equal on two distinct files, then those files are identical. This approach has been used in both real projects and in research efforts (for example `rysnc` [16] and LBFS [12]). A recent paper by Henson criticized this practice [8]. The present paper revisits the topic from an advocate's standpoint: we claim that compare-by-hash is completely reasonable, and we offer various arguments in support of this viewpoint in addition to addressing concerns raised by Henson.

Keywords: Cryptgraphic hash functions, distributed file systems, probability theory

1 Introduction

There is a well-known short-cut technique for testing two files for equality. The technique entails using a cryptographic hash function, such as SHA1 or RIPEMD-160, to compare the files: instead of comparing them byte-by-byte, we instead compare their hashes. If the hashes differ, then the files are certainly different; if the hashes agree, then the files are almost certainly the same. The motivation here is that the two files in question might live on opposite ends of a low-bandwidth network connection and therefore a substantial performance gain can be realized by sending a small (eg, 20 byte) hash digest rather than a large (eg, 20 megabyte) file. Even without this slow network connection, it is still a significant performance gain to compare hashes rather than compare files directly when the files live on the same disk: if we always keep the latest hash value for files of interest, we can quickly check the files for equality by once again comparing only a few bytes rather than reading through them byte-by-byte.

One well-known use of this technique is the file synchronizing tool called `rsync` [16]. This tool allows one to maintain two identical copies of a set of files on two separate machines. When updates are applied to some subset of the files on one machine, `rsync` copies those updates to the other machine. In order to determine which files have changed and which have not, `rsync` compares the hashes of respective files and if they match, assumes they have not changed since the last synchronization was performed.[1]

Another well-known use of compare-by-hash in the research community is the Low-Bandwidth File System (LBFS) [12]. The LBFS provides a fully-sychronized file system over low-bandwidth connections, once again using a cryptographic hash function (this time SHA1) to aid in determining when and where updates have to be distributed.

Cryptographic hash functions are found throughout cryptographic protocols as well: virtually every digital signature scheme requires that the message to be signed is first processed by a hash function, for example. Time-stamping mechanisms, some message-authentication protocols, and several widely-used public-key cryptosystems also use cryptographic hash functions.

A difference between the above scenarios is that usually in the compare-by-hash case (`rsync` and LBFS are our examples here) **there is no adversary.** Or at least the goal of the designers was not to provide security via the use of cryptographic hash functions. So in some sense using a cryptographic hash function is overkill: a hash function that is

[*] Department of Computer Science, 430 UCB, Boulder, Colorado 80309 USA. E-mail: jrblack@cs.colorado.edu WWW: www.cs.colorado.edu/∼jrblack/

[1] This is an oversimplification; `rsync` actually uses a lightweight "rolling" hash function in concert with a cryptographic hash function, MD4, on blocks of the files being compared. The simplified description will suffice for our purposes.

simply "good" at spreading distinct inputs randomly over some range should suffice. In the cryptographic examples, there *is* an adversary: indeed all the latter examples are taken from the cryptographic literature and the hash functions in these scenarios are specifically introduced in order to foil a would-be attacker.

In spite of the fact these functions are stronger than what is needed, a recent paper by Henson [8] criticizes the approach and gives a list of concerns. In this paper we will revisit some of the issues raised by Henson and argue that in most cases they are not of great concern. We will argue in favor of continuing to use compare-by-hash as a cheap and practical tool.

2 The Basics of Hash Functions

They are variously called "cryptographic hash functions," "collision-resistant hash functions," and "Universally One-Way Hash Functions," not to mention the various acronyms in common usage. Here we will just say "hash function" and assume we mean the cryptographic sort.

A hash function h accepts any string from $\{0,1\}^*$ and outputs some b-bit string called the "hash" or "digest." The most well-known hash functions are MD4, MD5 [14], RIPEMD-160 [6], and SHA1 [7]. The first two functions output digests of 128 bits, and the latter two output 160 bits.

Although it is not possible to rigorously define the security of these hash functions, we use the following (informal) definitions to capture the main goals.[2] A hash function h should be

- **Collision-Resistant:** it should be "computationally intractible" to find distinct inputs a and b such that $h(a) = h(b)$. Of course such a and b *must* exist given the infinite domain and finite range of h, but finding such a pair should be very hard.
- **Inversion-Resistant:** given any digest v, it should be "computationally intractible" to find an input a such that $h(a) = v$. This means that hashing a document should provide a "fingerprint" of that document without revealing anything about it.
- **Second-Preimage-Resistant:** given an input a and its digest $v = h(a)$, it should be "computationally intractible" to find a second input $b \neq a$ with $h(b) = v$. This is the condition necessary for secure digital signatures, which is the context in which hashing is most commonly employed cryptographically.

[2]For a proper discussion of these notions, along with a discussion of how they relate to one another, see [15].

In the context of compare-by-hash, collision-resistance is our main concern: if ever two distinct inputs produced the same digest, we would wrongly conclude that distinct objects were the same and this would result in an error whose severity is determined by the application; typically a file would be out-of-sync or a file system would become inconsistent.

Finding Collisions. If avoiding a collision (the event that two distinct inputs hash to the same output) is our main goal, we must determine how hard it is to find one. For a random function, this is given by the well-known "birthday bound." Roughly stated, if we are given a list of b-bit independent random strings, we expect to start seeing collisions after about $2^{b/2}$ strings have appeared. This means that MD4 and MD5 should begin showing collisions after we have seen about 2^{64} hash values, and RIPEMD-160 and SHA1 should begin showing collisions after about 2^{80} hash evaluations.

This is the expected number of hash evaluations before *any* pair of files collide. The probability that a *given* pair of files collide is much lower: 2^{-160} for SHA1, for example.

3 Compare-by-Hash: A Flawed Technique?

In her paper, Henson raises a number of issues which she deems important shortcomings of the compare-by-hash technique [8]. A central theme in Henson's paper is her opposition to the notion that digests can be treated as unique ids for blocks of data. She gives a list of arguments, often supported by numerical calculations, as to why this is a dangerous notion. In order to take an opposing viewpoint, we now visit several of her most prominent claims and examine them.

Incorrect Assumptions. One of Henson's most damning claims is her repeated assertion (cf. Sections 3 and 4.1) that in order for the outputs of a cryptographic hash function to be uniform and random, the *inputs* to the function must be random.[3] In fact, in Section 4.1, she claims that this supposedly-wrong assumption on the inputs is "the key insight into the weakness of compare-by-hash." She correctly points out that most file-system data are in fact *not* uniform over any domain. However, her assertion about hash functions is incorrect.

Cryptographic hash functions are designed to model "random functions." A random function from the set of all strings to the set of b-bit values can be thought of as an infinitely-long table where every possible string is listed in the left column, and for each row a random and independent b-bit number is listed in the right column. Obviously with this (fantasy) mapping, even highly-correlated distinct inputs will map to independent and random outputs. Of course any actual hash function we write down cannot have this property (since it is a static object the moment we write it down), most researchers agree that our best hash functions, like SHA1 and RIPEMD-160, do a very good job at mapping correlated inputs to uncorrelated outputs. (This is what's known as withstanding "differential cryptanalysis.")

If the assumption that inputs must be random is indeed the key insight into the weakness of compare-by-hash, as Henson claims, then we argue that perhaps compare-by-hash is not so weak after all.

Established Uses of Hashing. In section 3, Henson states that "compare-by-hash" sets a new precedent by relying solely on hash-value comparison in place of a direct comparison. While new precedents are not necessarily a bad thing, in this case it is simply untrue: one of the oldest uses of cryptographic hashing is (unsurprisingly) in cryptography. More specifically, they are universally used in digitial signature schemes where a document being digitally signed is first hashed with a cryptographic hash function, and then the signature is applied to the hash value. This is expressly done for performance reasons: applying a computationally-expensive digital signature to a large file would be prohibitive, but applying it to a short hash-function output is quite practical. The unscrupulous attacker now need only find a different (evil) file with the same hash value and this signature will appear valid for it as well. Therefore the security of the scheme rests squarely on the strength of the hash function used (in addition to the strength of the signature scheme itself), and the comparison of files is never performed: it is done entirely through the hash function. In fact, this could fairly be called "compare-by-hash" as well, and it has been accepted by the security community for decades.

[3]Technically, this statement cannot make sense since the input set is infinite and not compact and therefore cannot even have a probability measure.

Questionable Examples. Henson notes that if a massive collision-finding attempt for SHA1 were mounted by employing a large distributed brute-force attack using a compare-by-hash-based file system that also uses SHA1, collisions would not be detected. First, the example is a bit contrived: using a file-system based on SHA1 to look for collisions in SHA1 is a bit like testing NFS by mirroring it with the exact same implementation of NFS and then comparing to see if the results match. But even given this, it's still not clear that searching for SHA1 collisions using a SHA1-based compare-by-hash file system would present any problems. Van Oorschot and Wiener have described the best-known ways of doing parallel collision searching in this domain [17]. Let's use SHA1 in an example: each computer selects a (distinct, random) starting point x_0, and then iterates the hash function $x_i = \mathrm{SHA1}(x_{i-1})$, looking for "landmark" values that are stored in the file system. (These landmark values might be hash values with 40 leading zeros, for example. Under the assumption the hash outputs are uniform and random, we would expect to see such a point every 2^{40} iterations. We do this in order to avoid storing *all* hash values which for SHA1 would approximately require an expected 2^{85} bytes of storage, not including overhead for data structures for collision detection!) Since these landmark values would be written to the file system (along with other bookkeeping information all stored in some data structure), and since the number of blocks would be *far* less than 2^{80}, it's highly unlikely that even a SHA1-based compare-by-hash file system would have any difficulties at all.

The second example Henson gives is the VAL-1 hash function. The VAL-1 hash creates a publicly-known collision on two points, zero and one, and otherwise behaves like SHA1. Henson claims the VAL-1 hash has "almost identical probability of collision" as SHA1 and yet allows a user to make changes that would go undetected in a compare-by-hash-based file system. (The term "collision probability" is not defined anywhere in her paper.) Of course it is correct that the probability of collision is nearly the same between VAL-1 and SHA1 when the inputs are random (which itself is difficult to define as we mentioned above). But more to the point, VAL-1 does not have *security* equivalent to SHA1 by any normal notion of hash-function security in common use. The informal notion of collision-resistance given above dictates that it is intractible to find collisions in our hash function; in VAL-1 it is quite trivial since VAL-1(0) = VAL-1(1). (In the formal definitions for hash-function security, VAL-1 would fail to be secure as well.) Once again, the example is dubious due to the assumption that hash-function security rests solely on the distribution of digests over randomly-chosen inputs.

What is the Attack Model? Although Henson mentions correctness as the principal concern, she also mentions security. But throughout the paper it's unclear what the attack model is, exactly. More specifically, if we are worried about collisions, do we assume that our enemy is just bad luck, or is there an intelligent attacker trying to cause collisions?

If we're just concerned with bad luck, we have seen there is very little to worry about: the chance that two files will have the same hash (assuming once again that the hash function adequately approximates a random function) is about 2^{-160} for a 160-bit hash function like SHA1 or RIPEMD-160. If there is an active attacker *trying* to cause trouble, he must first somehow find a collision. This is a difficult proposition, but in the extremely unlikely case he succeeds, he may find blocks b_1 and b_2 that collide. In this case, he may freely cause problems by substituting b_1 for b_2 or vice-versa, and a compare-by-hash file system will not notice. However, finding this one collision (which we must emphasize has still not been done to-date) would not enable him to alter arbitrary blocks. If the method by which he found b_1 and b_2 was to try random blocks until he found a collision, it's highly unlikely that b_1 or b_2 are blocks of interest to him, that they have any real meaning, or that they even *exist* on the file system in question.

In either case, it would seem that compare-by-hash holds up well in both attack models.

The Insecurity of Cryptographic Objects. Henson spends a lot of time talking about the poor track-record of cryptographic algorithms (cf. Section 4.2), stating that the literature is "rife with examples of cryptosystems that turned out not to be nearly as secure as we thought." She further asserts that "history tells us that we should expect any popular cryptographic hash to be broken within a few years of its introduction." Although it is true that some algorithms are broken, this is by no means as routine as she implies.

The Davies-Meyers hash has been known since 1979, and the Matyas-Meyer-Oseas scheme since 1985 [11]. These have never been broken despite their being very well-known and well-analyzed [13, 2]. RIPEMD-160 and SHA1 were both published more recently (1995) but have still held up for more than just "a few years," since there is still no known practical attack against either algorithm. While it is true that there have been many published cryptographic ideas which were later broken, rarely have these schemes ever made it into FIPS, ANSI, or ISO standards. The vetting process usually weeds them out first. Probably the best example is the DES blockcipher which endured 25 years of analysis without any effective attacks being found.

Even when these "breaks" occur, they are often only of theoretical interest: they break because they fail to meet some very strict definition of security set forth by the cryptographic community, not because the attack could be used in any practical way by a malicious party.

Ask any security expert and he or she will tell you the same thing: if you want to subvert the security of a computer system, breaking the cryptography is almost never the expeditious route. It's highly more likely that there is some flaw in the system itself that is easier to discover and exploit than trying to find collisions in SHA1.

That said, there *have* been attacks on cryptographic hash functions over the past 10 years, and some very striking ones just in just the past few years. We briefly discuss these.

Recent Attacks on Cryptographic Hash Functions. Collisions in MD4 were found by Hans Dobbertin in 1996 [5]. However, MD4 is still used in `rsync` undoubtedly due to the arguments made above: there is typically no adversary, and just *happening* on a collision in MD4 is highly unlikely. MD4 was known to have shortcomings long before Dobbertin's attack, so its inventor (Ron Rivest) also produced the stronger hash function MD5. MD5 was also attacked by Dobbertin, with partial success, in 1996 [4]. However, full collisions in MD5 were not found until 2004 when Xiaoyun Wang shocked the community by announcing that she could find collsions in MD5 in a few hours on an IBM P690 [19]. Since that time, other researchers have refined her attack to produce collisions in MD5 in just a few minutes on a Pentium 4, on average [9, 1]. Clever application of this result has allowed various attacks: distinct X.509 certificates with the same MD5 digest [10], distinct postscript files with the same MD5 digest [3], and distinct Linux binaries with the same MD5 digest [1].

No inversion or second-preimage attacks have yet been found, but nonetheless cryptographers now consider MD5 to be compromised. SHA1 and RIPEMD-160 still have no published collisions, but many believe it is only a matter of time. Wang's latest attacks claim an attack complexity of around 2^{63}, well within the scope of a parallelized attack [18]. However, until the attack is implemented, we won't know for sure if her analysis is accurate.

Given all of this, we can now revisit the central question of this paper one final time: "is it safe to use cryptographic hash functions for compare-by-hash?" I argue that it is: even with MD4 (collisions can be found by hand in MD4!). This is, once again, because in the typical compare-by-hash setting there is no adversary. The event that two arbitrary distinct files will have the same MD4 hash is highly unlikely. And in the `rysnc` setting, comparisons are done between files only if they have the same filename; we don't compare all possible pairs of files across the filesystems. No birthday phenomenon exists here.

In the cryptographic arena, the issue is more complicated: there *is* an adversary and what can be done with the current attack technology is a topic currently generating much discussion. Where this ends up remains to be seen.

4 Conclusion

We conclude that it is certainly fine to use a 160-bit hash function like SHA1 or RIPEMD-160 with compare-by-hash. The chances of an accidental collision is about 2^{-160}. This is unimaginably small. You are roughly 2^{90} times more likely to win a U.S. state lottery *and* be struck by lightning simultaneously than you are to encounter this type of error in your file system.

In the adversarial model, the probability is higher but consider the following: it was estimated that general techniques and custom hardware could find collisions in a 128-bit hash function for 10,000,000 USD in an expected 24 days [17]. Given that this estimate was made in 1994 we could use Moore's law to extrapolate the cost in 2006 to be more like 80,000 USD. Since SHA1 has 32 more bits, we could rescale these estimates to be 80,000,000 USD and 2 years to find a collision in SHA1. This cost is far out of reach for anyone other than large corporations and governments. But if we are worried about adversarial users with lots of resources, it might make sense to use the new SHA-256 instead.

Of course, if someone is willing to spend 80,000,000 USD to break into your computer system, it would probably be better spent finding vulnerabilities in the operating system or on social engineering. And this approach would likely yield results in a lot less than 2 years.

References

[1] Black, J., Cochran, M., and Highland, T. A study of the MD5 attacks: Insights and improvements. In *Fast Software Encryption* (2006), Lecture Notes in Computer Science, Springer-Verlag.

[2] Black, J., Rogaway, P., and Shrimpton, T. Black-box analysis of the block-cipher-based hash-function constructions from PGV. In *Advances in Cryptology – CRYPTO '02* (2002), vol. 2442 of *Lecture Notes in Computer Science*, Springer-Verlag.

[3] Daum, M., and Lucks, S. Attacking hash functions by poisoned messages. Presented at the rump session of Eurocrypt '05.

[4] Dobbertin, H. Cryptanalysis of MD5 compress. Presented at the rump session of Eurocrypt '96.

[5] Dobbertin, H. Cryptanalysis of MD4. In *Fast Software Encryption* (1996), vol. 1039 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 53–69.

[6] Dobbertin, H., Bosselaers, A., and Preneel, B. Ripemd-160, a strengthened version of ripemd. In *3rd Workshop on Fast Software Encryption* (1996), vol. 1039 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 71–82.

[7] FIPS 180-1. Secure hash standard. NIST, US Dept. of Commerce, 1995.

[8] Henson, V. An analysis of compare-by-hash. In *HotOS: Hot Topics in Operating Systems* (2003), USENIX, pp. 13–18.

[9] Klima, V. Finding MD5 collisions on a notebook PC using multi-message modifications. In *International Scientific Conference Security and Protection of Information* (May 2005).

[10] Lenstra, A. K., and de Weger, B. On the possibility of constructing meaningful hash collisions for public keys. In *Information Security and Privacy, 10th Australasian Conference – ACISP 2005* (2005), vol. 3574 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 267–279.

[11] Matyas, S., Meyer, C., and Oseas, J. Generating strong one-way functions with cryptographic algorithms. *IBM Technical Disclosure Bulletin 27*, 10a (1985), 5658–5659.

[12] Muthitacharoen, A., Chen, B., and Mazières, D. A low-bandwidth network file system. In *SOSP: ACM Symposium on Operating System Principles* (2001), pp. 174–187.

[13] Preneel, B., Govaerts, R., and Vandewalle, J. Hash functions based on block ciphers: A synthetic approach. In *Advances in Cryptology – CRYPTO '93* (1994), Lecture Notes in Computer Science, Springer-Verlag, pp. 368–378.

[14] RFC 1321. The MD5 message digest algorithm. IETF RFC-1321, R. Rivest, 1992.

[15] Rogaway, P., and Shrimpton, T. Cryptographic hash-function basics: Definitions, implications, and separations for preimage resistance, second-preimage resistance, and collision resistance. In *FSE: Fast Software Encryption* (2004), Springer-Verlag, pp. 371–388.

[16] Tridgell, A. Efficient algorithms for sorting and synchronization, Apr. 2000. PhD thesis, Australian National University.

[17] van Oorschot, P., and Wiener, M. Parallel collision search with cryptanalytic applications. *Journal of Cryptology 12*, 1 (1999), 1–28. Earlier version in ACM CCS '94.

[18] Wang, X., Yao, A., and Yao, F. Recent improvements in finding collisions in sha-1. Presented at the rump session of CRYPTO '05 by Adi Shamir.

[19] Wang, X., and Yu, H. How to break MD5 and other hash functions. In *EUROCRYPT* (2005), vol. 3494 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 19–35.

A A Mathematical Note

This appendix is related to the paper, but not essential to the arguments made.

A further issue with Henson's paper should be pointed out for purposes of clarity. Many of Henson's claims are supported by calculation. However these calculations are sometimes not quite right: in section 3, she claims that the probability of one or more collisions among n outputs from a b-bit cryptographic hash function is $1 - (1 - 2^{-b})^n$. (Here we model the b-bit outputs as independent uniform random strings.) This is incorrect. In fact it greatly *underestimates* the collision probability. While it is true the probability that two b-bit random strings will not collide is $(1 - 2^{-b})$, we cannot raise this to the n and expect this to be the number of non-collisions among n total outputs since it neglects to count collisions across already-selected pairs.

The probability that n outputs of b-bits will contain a collision is $C(2^b, n) = 1 - (2^b - 1)/2^b \times (2^b - 2)/2^b \times \cdots \times (2^b - n + 1)/2^b$. It can be shown that, when $1 \leq n \leq 2^{(b+1)/2}$ we have $0.316n(n-1)/2^b \leq C(2^b, n) \leq 0.5n(n-1)/2^b$. This formula shows that the probability of a collision is *very* unlikely when n is well below the square root of 2^b, and then grows dramatically as n approaches this number. Indeed, it can be shown that the expected number of outputs needed before a collision occurs is $n \approx \sqrt{2^{b-1}\pi}$.