An Analysis of Compare-by-hash

Val Henson
Sun Microsystems
vhenson@eng.sun.com

Abstract

Recent research has produced a new and perhaps dangerous technique for uniquely identifying blocks that I will call *compare-by-hash*. Using this technique, we decide whether two blocks are identical to each other by comparing their hash values, using a collision-resistant hash such as SHA-1[5]. If the hash values match, we assume the blocks are identical without further ado. Users of compare-by-hash argue that this assumption is warranted because the chance of a hash collision between any two randomly generated blocks is estimated to be many orders of magnitude smaller than the chance of many kinds of hardware errors. Further analysis shows that this approach is not as risk-free as it seems at first glance.

1 Introduction

Compare-by-hash is a technique that trades on the insight that applications frequently read or write data that is identical to already existing data. Rather than read or write the data a second time to the disk, network, or memory, we should use the instance of the data that we already have. Using a collision-resistant hash, we can quickly determine with a high degree of accuracy whether two blocks are identical by comparing only their hashes and not their contents. After making a few assumptions, we can estimate that the chance of a hash collision is much lower than the chance of a hardware error, and so many feel comfortable neglecting the possibility of a hash collision.

Compare-by-hash is accepted by some computer scientists and has been implemented in several different projects: rsync[14], a utility for synchronizing files, LBFS[4], a distributed file system, Stanford's virtual computer migration project[9], Venti[6], a block archival system, Pastiche[3], an automated backup system, and OpenCM[11], a configuration management system. However, I believe some publications overstate the acceptance of compare-by-hash, claiming that it is "customary"[3] or a "widely-accepted practice"[4] to assume hashes never collide in this context. An informal survey of my colleagues reveals that many computer scientists are still either unaware of compare-by-hash or disagree with the technique strongly. Since adoption of compare-by-hash has the potential to change the face of operating systems design and implementation, it should be the subject of more criticism and peer review before being accepted as a general purpose computing technique for critical applications.

In this position paper, I hope to begin an in-depth discussion of compare-by-hash. Section 2 reviews the traditional uses of hashing, followed by a more detailed description of compare-by-hash in Section 3. Section 4 will raise some questions about the use of compare-by-hash as a general-purpose technique. Section 5 will propose some alternatives to compare-by-hash, and Section 6 will summarize my findings and make recommendations.

2 Traditional applications of hashing

The review in this section may seem tedious and unnecessary, but I believe that a clear understanding of how hashing has been used in the past is necessary to understand how compare-by-hash differs.

A **hash function** maps a variable length input string to fixed length output string — its **hash value**, or **hash** for short. If the input is longer than the output, then some inputs must map to the same output — a **hash collision**. Comparing the hash values for two inputs can give us one of two answers: the inputs are definitely not the same, or there is a possibility that they are the same. Hashing as we know it is used for performance improvement, error checking, authentication, and encryption. One example of a performance improvement is the common hash table, which uses a hash function to index into the correct bucket in the hash table, followed by comparing each element in the bucket to find a match. In error checking, hashes (checksums, message digests, etc.) are used to detect errors caused by either hardware or software. Examples are TCP checksums, ECC memory, and MD5 checksums on

downloaded files[1]. In this case, the hash provides additional assurance that the data we received is correct. Finally, hashes are used to authenticate messages. In this case, we are trying to protect the original input from tampering, and we select a hash that is strong enough to make malicious attack infeasible or unprofitable.

3 Compare-by-hash in detail

Compare-by-hash is a technique used when the payoff of discovering identical blocks is worth the computational cost of computing the hash of a block. In compare-by-hash, we assume hash collisions never occur, so we can treat the hash of a block as a unique id and compare only the hashes of blocks rather than the contents of blocks. For example, we can use compare-by-hash to reduce bandwidth usage. Before sending a block, the sender first transmits the hash of the block to the receiver. The receiver checks to see if it has a local block with the same hash value. If it does, it assumes that it is the same block as the sender's, without actually comparing the two input blocks. In the case of a 4096 byte block and a 160 bit hash value, this system can reduce network traffic from 4096 bytes to 20 bytes, or about a 99.5% savings in bandwidth.

This is an incredible savings! The cost, of course, is the risk of a hash collision. We can reduce that risk by choosing a collision-resistant hash. From a cryptographic point of view, collision resistance means that it is difficult to find two inputs that hash to the same output. By implication, the range of hash values must be large enough that a brute-force attack to find collisions is "difficult."[2] Cryptologists have given us several algorithms that appear to have this property, although so far, only SHA-1 and RIPEMD-160 have stood up to careful analysis[8].

With a few assumptions, we can arrive at an estimate for the risk of a hash collision. We assume that the inputs to the hash function are random and uniformly distributed, and the output of the hash function is also random and uniformly distributed. Let n be the number of input blocks, and let b be the number of bits in the hash output. As a function of the number of input blocks, n, the probability that we will encounter one or more collisions is $1-(1-2^{-b})^n$. This is a difficult number to calculate when b is 160, but we can use the "birthday paradox"[3] to calculate how many inputs will give us a 50% chance of finding a collision. For a 160-bit output, we will need about $2^{160/2}$ or 2^{80} inputs to have a 50% chance of a collision. Put another way, we expect with about 48 nines ($1-2^{-160}$) of certainty that any two randomly chosen inputs will not collide, whereas empirical measurements tell us we only have perhaps 8 or 9 nines of certainty that we will not encounter an undetected TCP error when we transmit the block[13]. In the face of much larger sources of potential error, the error added by compare-by-hash appears to be negligible.

Now that we've described compare-by-hash in more detail, it should be clear how compare-by-hash and traditional hashing differ: No known previous uses of hashing skip the step of directly comparing the inputs for performance reasons. The only case in which we do skip that step is authentication, because we can't compare the inputs directly due to the lack of a secure channel. Compare-by-hash sets a new precedent and so does not yet enjoy the acceptance of established uses of hashing.

4 Questions about compare-by-hash

What appears to be a fall of manna from heaven should be examined a little more closely before compare-by-hash is accepted into the computer scientist's tricks of the trade. In the following section, I will re-examine the assumptions we made earlier when justifying the use of compare-by-hash.

4.1 Randomness of input

In Section 3, we calculated the probability of a hash collision under the assumption that our inputs were random and uniformly distributed. While this assumption simplifies the math, it is also **wrong**.

Real data is not random, unless all applications produce random data. This may seem like a trivial and facile statement, but it is actually the key insight into the weakness of compare-by-hash. If real data were actually random, each possible input block would be equally likely to occur, whereas in real data, input blocks that contain only ASCII characters or begin with an ELF header are more common than in random data. Knowing that real data isn't

[1] MD5 checksums are designed to detect intentional tampering as well.

[2] A **cryptographically secure hash** is defined as a hash with no known method better than brute force for finding collisions.

[3] The "birthday paradox" is best illustrated by the question, "How many people do you need in a room to have a 50% or greater chance that two of them have the same birthday?" The answer is 23 (assuming that birthdays are uniformly distributed and neglecting leap years). This is easier to understand if you realize that there are $23 \times (22/2) = 253$ different *pairs* of people in the room.

random, can we think of some cases where it is non-random in an interesting way?

Consider an application, let's call it SHA1@home, that attempts to find a collision in the SHA-1 hash function. SHA1@home is a distributed application, so it runs many instances in parallel on many machines, using a distributed file system to share data when necessary. When two inputs are found that hash to the same value, one program reads and compares both input blocks to find out if they differ. If the file system uses compare-by-hash with SHA-1 and the same block size as the inputs for SHA1@home, this application will be unable to detect a collision, ever. For example, if SHA1@home used a 2KB block size, it would run incorrectly if it used LBFS as the underlying file system[4].

This is only one very crude, very simple example of an entire class of applications that are very useful, especially to cryptanalysts. In their 1998 paper, Chabaud and Joux implemented several programs designed specifically to find collisions in various hashing algorithms, including SHA-0 and several relatives. They end by hinting at avenues of research for attacking SHA-1[2]. Somewhat ironically, this paper is referenced by one of the papers using compare-by-hash[9].

4.2 Cryptographic hashes — one size fits all?

Collision-resistant hashes were originally developed for use in cryptosystems. Is a hash intended for cryptography also good for use in systems with different characteristics?

Cryptographic hashes are short-lived. Data is forever, secrecy is not. The literature is rife with examples of cryptosystems that turned out to not be nearly as secure as we thought. Weakness are frequently discovered within a few years of a cryptographic hash's introduction[2, 8, 10]. On the other hand, lifetimes of operating systems, file systems, and file transfer protocols are frequently measured in decades. Solaris, FFS, and ftp come to mind immediately. Cryptologists choose algorithms based on how long they want to keep their data secure, while computer scientists should choose their algorithms based on how long they want to keep their data, period. (Cryptologists may desire to keep data secure for decades, but most would not expect their current algorithms to actually accomplish this goal.)

[4]LBFS uses variable sized blocks, but has minimum block size of 2KB to avoid pathologically small block sizes[4]

Obsolecence can occur overnight. A related consideration is how quickly obsolescence occurs for cryptosystems. In operating systems, we are used to systems slowing and gracefully obsolescing over a period of years. Cryptosystems can go from state-of-the-art to completely useless overnight.

Obsolescence is inevitable. Large governments, corporations, and scientists all have a huge incentive to analyze and break cryptographic hashes. We have no proof that any particular hash, much less SHA-1, is "unbreakable." At the same time, history tells us that we should expect any popular cryptographic hash to be broken within a few years of its introduction. If anyone had built a distributed file system using compare-by-hash and MD4, it would already be unusable today, due to known attacks that take seconds to find a collision using a personal computer. MD5 appears to be well on its way to unusability as well[8].

Upgrade strategy required. Given that our hash algorithms will be obsolete within a few years, systems using compare-by-hash need to have a concrete upgrade plan for what happens when anyone with a PC can generate a hash collision. Upgrade will be more difficult if any hash collisions have occurred, because part of your data will now be corrupted, possibly a very important part of your data.

4.3 Silent, deterministic, hard-to-fix errors

Ordinarily, anyone who discovered two inputs that hash to the same SHA-1 output would become world-famous overnight. On a system using compare-by-hash, that person would instead just silently read or overwrite the wrong data (which is more than a bug, it's a security hole). To understand why silent errors are so bad, think about silent disk corruption. Sometimes the corruption goes undetected until long after the last backup with correct data has been destroyed.

In addition, any two inputs that hash to the same value will always be treated incorrectly, whereas most hardware errors are transitory and data-independent. Redundant disks or servers provide no protection against data-dependent, deterministic errors. To avoid this, we could add a random seed every time we compute the hash, but we won't save anything except in the most extreme cases if we have to recompute hashes on every candidate local block every time we compare a block.

Once a hash collision has been found and a demonstrably buggy test program created using the collid-

ing inputs, how will you fix the bug? Usually, the response to a test program that demonstrates a bug in the system is to fix the bug. In this case, the underlying algorithm is the bug.

4.4 Comparing probabilities

One of the primary arguments for compare-by-hash is a simple comparison of the probability of a hash collision (very low) and the probability of some common hardware error (also low but much higher). To show that we cannot directly compare the probability of a deterministic, data-dependent error with the probability of nondeterministic, data-independent error, let's construct a hash function that has the same collision probability as SHA-1 but, when used in compare-by-hash, will be a far more common source of error than any normal hardware error.

Define VAL-1(x) as follows:

$$\text{VAL-1}(x) = \begin{cases} x > 0 & : \quad \text{SHA-1}(x) \\ x = 0 & : \quad \text{SHA-1}(1) \end{cases}$$

In other words, VAL-1 is SHA-1 except that the first two inputs map to the same output. This function has an almost identical probability of collision as SHA-1, but it is completely unsuitable for use in compare-by-hash. The point of this example is not that bad hash functions will result in errors, but that we can't directly compare the probability of a hash collision with the probability of a hardware error. If we could, VAL-1 and SHA-1 would be equally good candidates for compare-by-hash. The relationship between the probability of a hash collision and the probability of a hardware error must be more complicated than a straightforward comparison can reveal.

4.5 Software and reliability

On a more philosophical note, should software improve on hardware reliability or should programmers accept hardware reliability as an upper bound on total system reliability? What would we think of a file system that had a race condition that was triggered less often than disk I/O errors? What if it lost files only slightly less often than users accidentally deleted them? Once we start playing the game of error relativity, where do we stop? Current software practices suggest that most programmers believe software should improve reliability — hence we have TCP checksums, asserts for impossible hardware conditions, and handling of I/O errors. For example, the empirically observed rate of undetected errors in TCP packets is about 0.0000005%[13]. We could dramatically improve that rate by sending both the block and its SHA-1 hash, or we could slightly worsen that rate by sending only the hash.

4.6 When is compare-by-hash appropriate?

Taking all this into account, when is it reasonable to use compare-by-hash? For one, users of software should know when they are getting best effort and when they are getting correctness. When using rsync, the user knows that there is a tiny but real possibility of an incorrect target file (in rsync's case, the user has only to read the man page). When using a file system, or incurring a page fault, users expect to get exactly the data they wrote, all the time. Another consideration is whether other users share the "address space" produced by compare-by-hash. If only trusted users write data to the system, they don't have to worry about maliciously generated collisions and can avoid known collisions. By these standards, rsync is an appropriate use of compare-by-hash, whereas LBFS, Venti, Pastiche, and Stanford's virtual machine migration are not.

5 Alternatives to compare-by-hash

The alternatives to compare-by-hash can be summarized as "Keep some state!" Compare-by-hash attempts to establish similarities between two unknown sets of blocks. If we keep track of which blocks we are sure are identical (because we directly compared them), we don't have to guess. Unfortunately, keeping state is hard. Part of the popularity of compare-by-hash is undoubtably due to its ease of implementation compared to a stateful solution. However, simplicity of implementation should not come at the cost of correctness.

One of the applications of compare-by-hash is reducing network bandwidth used by distributed file systems. To accomplish nearly the same effect, we can resolve to only send any particular block over the link once, keeping sent and received data in a cache in both sender and receiver. Before sending a block, the sender checks to see if it has already sent the block and if so, sends the block id rather than the block itself. This idea is proposed by Spring and Wetherall in [12]. We might also agree in advance on certain universal block ids, for example, block id 0 is always the zero block of length 4096 bytes. The initial start-up cost is higher, depending on the degree of actually shared blocks between the two machines, but after cache warm-up, performance should be quite similar to compare-by-hash.

In combination with an intelligent blocking tech-

nique, such as Rabin fingerprints[7], which divide up blocks at "anchor" points (patterns in the input) rather than at fixed intervals, we can experiment with byte and block level differencing techniques that require similar amounts of computation time as computing cryptographic hashes. Using fingerprints to determine block boundaries allows us to more easily detect insertions and deletions within blocks.

Compression may still have more mileage left in it, since we are willing to trade off large amounts of computation for reduced bandwidth. We might try compressing with several different algorithms optimized for different inputs.

5.1 Existence proof: Rsync vs. BitKeeper

As an example of a system that improves on compare-by-hash while retaining correctness, compare rsync and BitKeeper[1], a commercial source configuration management tool. They both solve the problem of keeping several source code trees in sync. (We will ignore the unrelated features of BitKeeper, such as versioning, in this comparison.) Rsync is stateless; it has no a priori knowledge of the relationship between two source code trees. It uses compare-by-hash to determine which blocks are different between the two trees and sends only the blocks with different hashes. BitKeeper keeps state about each file under source control and knows what changes have been made since the last time each tree was synchronized. When synchronizing, it sends only the differences since the last synchronization occurred, in compressed form. In comparison to rsync, BitKeeper provides similar and sometimes better bandwidth usage when simply synchronizing two trees without resorting to compare-by-hash. Improvements BitKeeper provides over rsync include elimination of reverse updates (synchronizing in the wrong direction and losing your changes), automerging algorithms optimized for source code (so trees can be updated in parallel and then synchronized), and intelligent handling of metadata operations such as renaming of files (which rsync sees as deletion and creation of files).

With a little more programming effort, we can get the bandwidth reduction promised by compare-by-hash without sacrificing correctness and at the same time adding functionality. Compare-by-hash still has applications in areas where statelessness and low bandwidth are more important than correctness of data referenced, and users are aware of the risk they are taking, as in rsync.

6 Conclusion

Use of compare-by-hash is justified by mathematical calculations based on assumptions that range from unproven to demonstrably wrong. The short lifetime and fast transition into obsolescence of cryptographic hashes makes them unsuitable for use in long-lived systems. When hash collisions do occur, they cause silent errors and bugs that are difficult to repair. What should worry computer scientists the most about compare-by-hash is that real people are running real workloads that will execute incorrectly on systems using compare-by-hash. Perhaps research would be better directed towards alternatives to or improvements on compare-by-hash that avoid the problems described. At the very least, future research using compare-by-hash should include a more careful analysis of the risk of hash collisions.

7 Acknowledgments

Many people joined in on (both sides of) the discussion that led to this paper and provided helpful comments on drafts, including Jonathan Adams, Matt Ahrens, Jeff Bonwick, Bryan Cantrill, Miguel Castro, Whit Diffie, Marius Eriksen, Barry Hayes, Richard Henderson, Larry McVoy, Dave Powell, Bart Smaalders, Niraj Tolia, Vernor Vinge, and Cynthia Wong.

References

[1] Bitmover, Inc. Bitkeeper - the scalable distributed software configuration management system. http://www.bitkeeper.com.

[2] Florent Chabuad and Antoine Joux. Differential collisions in SHA-0. In *Proceedings of CRYPTO '98, 18th Annual International Cryptology Conference*, pages 56–71, 1998.

[3] Landon P. Cox, Christoper D. Murray, and Brian D. Noble. Pastiche: Making backup cheap and easy. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, 2002.

[4] Athicha Muthitacharoen, Benjie Chen, and David Maziéres. A low-bandwidth network file system. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, 2001.

[5] National Institute of Standards and Technology. *FIPS Publication 180–1: Secure Hash Standard*, 1995.

[6] Sean Quinlan and Sean Dorward. Venti: a new approach to archival storage. In *Proceedings of the FAST 2002 Conference on File and Storage Technologies*, 2002.

[7] M. O. Rabin. Fingerprinting by random polynomials. Technical Report TR–15–81, Center for Research in Computer Technology, Harvard University, 1981.

[8] B. Van Rompay, B. Preneel, and J. Vandewalle. On the security of dedicated hash functions. In *19th Symposium on Information Theory in the Benelux*, 1998.

[9] Constantine P. Sapuntzakis, Ramesh Chandra, Ben Pfaff, Jim Chow, Monica S. Lam, and Mendel Rosenblum. Optimizing the migration of virtual computers. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, 2002.

[10] Bruce Schneier. *Applied Cryptography*. John Wiley & Sons, Inc., second edition, 1996.

[11] Jonathan S. Shapiro and John Vanderburgh. CPCMS: A configuration management system based on cryptographic names. In *Proceedings of the 2002 USENIX Technical Conference, FREENIX Track*, 2002.

[12] Neil T. Spring and David Wetherall. A protocol independent technique for eliminating redundant network traffic. In *Proceedings of the 2000 ACM SIGCOMM Conference*, 2000.

[13] Jonathan Stone and Craig Partridge. When the CRC and TCP checksum disagree. In *Proceedings of the 2000 ACM SIGCOMM Conference*, 2000.

[14] Andrew Tridgell. *Efficient Algorithms for Sorting and Synchronization*. PhD thesis, The Australian National University, 1999.