SIGINFO

The Tricky Cryptographic Hash Function

SIMSON L. GARFINKEL

Simson L. Garfinkel is a Senior Computer Scientist at the US Census Bureau and a researcher in digital forensics and usability. He recently published *The Computer Book*, a coffee table book about the history of computing. sigmail@simson.net

Cryptographic hash functions are one of the building blocks of modern computing systems. Although they were originally developed for signing digital signatures with public key cryptography, they have found uses in digital forensics, digital timestamping, and cryptocurrency schemes like Bitcoin.

Cryptographic hash functions like MD5, SHA-1, and BLAKE3 are widely used and appreciated by programmers, end users, and even lawyers! Nevertheless, I'll start off this column with a basic description of what hash functions are and the hash functions that are used today. Then I'll delve back to the first references to them that I've been able to find and give a bit of their history. I'll briefly touch on their uses in cryptography and then discuss how they also found use in digital forensics. I'll end with a puzzle from Stuart Haber, one of the co-inventors of the blockchain concept. Unless otherwise noted, all of the timing runs were performed on my Mac mini (vintage 2018) with a six-core Intel Core i5 processor running at 3 GHz. The hashing was done with OpenSSL 1.1.1d, compiled September 10, 2019, that ships with the Anaconda Python distribution.

Hash Functions

Hash functions take a sequence of bytes of any length and crunch it down to a block of seemingly random bits and a constant length. This block is typically called the *hash*, taken from the popular dish that involves chopping up food and then cooking it together.

Hash functions are widely used in computer science—they are the basis of the Python dictionary, for example. The basic idea of hashing was invented by IBM scientist Peter Luhn back in the 1950s as a technique to help speed up searching for words in text [1].

Cryptographic hash functions are fundamentally different from the hash functions that Luhn developed. For starters, their output is much larger. Python's hash function takes a string and returns an `int`—that is, 32 or 64 bits—which then becomes an index into an array (modulo the size of the array). Cryptographic hash functions return more than a hundred bits, each (ideally) with an equal and independent probability of being a 0 or a 1, which is then used as a kind of placeholder for the object itself. Writing in RFC 1186 back in 1990 about his MD4 algorithm, Ron Rivest stated: "The algorithm takes as input an input message of arbitrary length and produces as output a 128-bit 'fingerprint' or 'message digest' of the input. It is conjectured that it is computationally infeasible to produce two messages having the same message digest, or to produce any message having a given prespecified target message digest."

The field of cryptographic hash functions has evolved considerably since 1990. Today we say that these functions should have several properties. First, it should be computationally infeasible to find a sequence of bytes that has a specific hash, called *pre-image resistance*. It should also be infeasible to find a second sequence m_2 that has the same hash as a first sequence m_1, called *second pre-image resistance*, or to find any two objects that have the same hash, called *collision resistance*. Finally, the output of the hash function should be indistinguishable from a random number generator. That is, there should be no way to predict the output of the hash from its input other than by running the actual hash function. This is called *pseudo-randomness*.

Cryptographic hash functions were first described in detail by Ralph Merkle in his 1979 PhD thesis [2], published just a few years after Diffie and Hellman introduced the world to public key cryptography and Rivest, Shamir, and Adleman disclosed the public key system that has memorialized their initials. Merkle called the functions "one-way hash functions," because it was easy to take a message and find its corresponding hash, but "effectively impossible"—or at least "computationally infeasible"— to take a hash and find a corresponding message. The RSA cryptosystem can't sign a number larger than the product of the prime numbers p and q—which today is typically a few thousand bits. Given a one-way hash, Merkle wrote, the newfangled digital signature schemes could be used to sign a message of any length: simply hash the message first, then sign the hash.

The idea of hashing a message and then signing the hash is standard operating procedure, but back in 1979 this was brand new stuff. What I find so enchanting about Merkle's PhD thesis is the combination of wonder, excitement, and amazement it conveys. Merkle's words help me to understand what it was like to live in a world where public key cryptography was new and nobody really knew how to use it or even quite what to do with it.

Today we know lots of things that you can do with hash functions—even without public key technology. In his PhD thesis, Merkle shows how it's possible to create digital signatures with just a one-way hash function. We now call these *Merkle signatures*. The critical insight is that you can take a secret message (call it M0) and hash it (call that H0). If you hand-deliver H0 to a friend today, you can send an authenticated message to your friend at some point in the future by sending M0. Your friend can verify the authenticity of M0 by hashing it and producing H0. In his thesis, Merkle credits this original idea to Leslie Lamport, as described in Diffie and Hellman's original "New Directions" [3] article, although Merkle notes that the scheme is much more efficient using cryptographic hash functions.

Of course, just being able to send a 0 by itself is not useful. So instead of giving your friend just H0, you give the friend H0 and H1 (which is the hash of M1). Now you can send one bit of authenticated information—either a zero or a one—by choosing to reveal either M0 or M1. Give your friend 256 different H0s and 256 different H1s, and you can now send 256 bits of digitally signed data. The disadvantage of this scheme is that each signature block can only be used once, so it's not tremendously efficient (although there are ways around this problem as well). The advantage of Merkle signatures is that they are very fast to compute, and it is widely thought that they are resistant to cracking by quantum computers, should such machines ever become practical.

If you want to sign 10 documents at the same time, you can compute the hash of each document (call that DH0 through DH9), then concatenate all of these hashes together, hash the resulting block (call that DHH), and sign that. You can prove the signature of any document by giving someone that document, the hashes for the other nine documents, and the signature for DHH: the verifier recomputes the missing document hash, uses DH0 through DH9 to compute DHH, and verifies that. This approach and the corresponding data structure, when extended to multiple levels, is now called a Merkle Tree.

The Rise and Fall of Many Hash Functions

The first widely used cryptographic hash function was MD2, developed by Rivest for use in an early secure email system. The source code for MD2, dated October 1, 1988, appears in RFC 1115, one of the early RFCs describing a system for sending encrypted messages over the Internet. This system was never widely adopted, but its ideas and data formats were quite influential.

Although no practical attack on MD2 was ever published, researchers started publishing theoretical attacks against it in 2004. Support for MD2 was removed from the popular OpenSSL cryptographic toolkit in 2009. But the real problem with MD2 wasn't its security but its speed: MD2 is an extraordinarily slow algorithm. Even on my 2018 Mac mini, Rivest's 1988 code takes 43 seconds to hash 256 MiB of data. Imagine how slowly it ran back in 1988!

Rivest went back to the drawing board. MD3 didn't make it out the door, but MD4 was released and appears in RFC 1186 (October 1990). Flaws were soon discovered in MD4 and it was not widely used. In 1991, Rivest invented MD5; the algorithm was published by the Internet Engineering Task Force (IETF) in April 1992 as RFC 1321.

MD5 is more than a hundred times faster than MD2; on my Mac mini, OpenSSL's MD5 implementation hashes that same 256 MiB file in just 0.37 seconds. Like MD2, MD5 also produces a 128-bit hash.

MD5 is still in use today, but it should no longer be used because it is now relatively straightforward to generate two blocks of data that have the same MD5 hash. That is, MD5 no longer has collision resistance. The first MD5 collision was demonstrated back in 2004; the Wikipedia article on MD5 has a nice write-up about how to create two documents that have an MD5 collision.

On the other hand, there is still no publicly known attack on MD5 that will let you find a block of data with a specific MD5 hash—that is, it still is publicly considered to have *pre-image resistance*. Nevertheless, MD5 is not to be trusted. For example, Amazon's Simple Storage Service (S3) still uses the MD5 algorithm for the "ETag" value that lets users check the integrity of uploaded files. The idea is that your software can compute the MD5 of a file, upload the file to S3, and then check the file's ETag to see if the value is the same. Although this works in practice, if you happen

to upload two files that have the same MD5, Amazon will happily give them both the same ETag.

In digital forensics, it's common to use file hashes to search a computer for *files of interest*, a broad term that includes stolen corporate documents, child sexual abuse materials, and other kinds of documents sought by investigators. Typically, an organization looking for materials will distribute a list of hashes for such files to investigators in the field. The investigators then run a program that hashes every file on a suspect's laptop and sees if any of those files has a hash that matches the list. If there's a match, then the suspect presumably has the file of interest. MD5 is still used in this application: after a collision is found, the investigator then looks at the matching file to see if it is in fact the file being sought.

Nevertheless, even in these applications, I try to avoid using MD5. That's because there are many articles on the Internet telling people not to use MD5 because it is not secure. I just don't think that it's a good use of one's time to argue that it's acceptable to use MD5 for some applications but not others.

Another hash function that is in wide use is SHA-1, the Secure Hash Algorithm, adopted by the National Institute of Standards and Technology in 1995. SHA-1 produces a 160-bit hash. Even though concerns about SHA-1 were raised within a few years of its being published, the National Institute of Standards and Technology (NIST) didn't formally recommend that we stop using SHA-1 until 2006. Eleven years later, Google published two PDFs that had identical SHA-1 hash values but render differently [4]. The two files are each 422,435 bytes long and differ in 62 bytes. They also look visually similar, except that one has a blue banner across the top while the other has a red banner.

As Andrew Tannenbaum once said, the nice thing about standards is that there are so many of them to choose from. Realizing that SHA-1 was likely to be compromised, in 2001 NIST revised the Secure Hash standard to allow for more rounds of computation and longer hash values, also called *residues*. Collectively called SHA-2, these revised algorithms include SHA-256, SHA-384, and SHA-512. In 2006 NIST initiated a competition for a new Secure Hash Algorithm. Nine years later NIST declared that an algorithm called Keccak would be adopted as SHA-3. This new algorithm is based on a fundamentally new mathematical approach called a sponge construction, in which input data are absorbed and then the hashed value is squeezed out. For details about these algorithms, as well as the multiple controversies surrounding their adoption, I refer you to the Wikipedia pages for SHA-1, SHA-2 and SHA-3.

It used to be the case that MD5 was dramatically faster than SHA-1, which was faster than SHA-256 (the 256-bit version of SHA-1), which was faster than SHA-512. That's no longer the case, in part due to better implementations and in part due to the fact that we're now running on 64-bit processors. In Table 1, I present the times to hash a 1 GiB file with several of the algorithms I mentioned above.

Bits	128	160	256	384	512
Family					
MD5	1.45				
SHA-1		1.03			
SHA-2			2.18	1.48	1.48
SHA-3			2.65	3.45	4.90

Table 1: Time in seconds to hash 1 GiB using OpenSSL 1.1.1d on the author's 2018 Mac mini

Hashing in Digital Forensics

Beyond searching for contraband, over the past three decades, digital forensics researchers have developed approaches to use cryptographic hashes for authenticating evidence, searching for file fragments, and even gauging file similarity. We can now even search a hard drive for contraband data in less time than it takes to read the hard drive's contents! These more sophisticated approaches have yet to be widely adopted, showing the difficulty of moving techniques from the lab to the field.

There are many definitions of digital forensics, but most of them link it to the recovery and analysis of digital information. Digital forensics techniques have many uses, including data recovery, event reconstruction, malware analysis, and even analyzing systems for the leakage of personal information. One of the best-known uses of digital forensics, though, is taking data from devices that were used by criminals and using that data as evidence in a court of law.

One of the early uses of cryptographic hash functions in digital forensics was to certify that the copy of a hard drive made by an investigator had not been changed after it was acquired. Forensics software would make a copy of the hard drive, called a *disk image*, and then compute the cryptographic hash of the disk image. The investigator would then write this hash in ink into their investigator's notebook. Although the computer scientist in me wishes that the early programs would have then signed this hash with a private key, this pen-and-paper record provided sufficient validation for US courts.

The fact that you could make two, five, or even 50 disk images of the same hard drive and they would all have the same hash engendered a lot of confidence in this basic digital forensics technique: a hashed disk image became the gold standard of digital evidence preservation and created the assumption that the data in the disk image was unchanged since the disk was seized from the suspect. Of course, this assumption was wrong: a crooked officer could easily have planted the incriminating evidence on

the hard drive *before it was imaged*. Such malfeasance is rare, fortunately, and there are other forensic techniques that can both detect and defend against such behavior.

These days, hashes are still used to establish that data taken from a digital device hasn't been altered since it was originally captured. However, the ability to repeatedly reimage a device and consistently get the same hash is quickly fading. When a modern operating system deletes a file, it can tell a solid state drive (SSD) to proactively erase the associated flash storage pages using the ATA TRIM command (called UNMAP in the SCSI command set). The drive doesn't immediately erase the page, but it may do so in the future. If the disk is imaged before the pages are erased, the disk image will contain the blocks' now-deleted data. But if the disk is left turned on, it may eventually erase these blocks. If you image the SSD a second time, then the blocks that were erased will now read as zeros, and the second image will have a different hash than the first. It is also increasingly difficult to get a "disk image" of a cell phone, as the data on many cell phones is accessed through an API, rather than by mounting the cell phone's internal storage. Such file collections are sometimes called "logical images."

If you use a hash that is 160 bits long, you can split it into six numbers of 25 bits each (throw out the remaining 10). If you have an array of 2^{25} bits, you can store information relating to that hash by setting the six indices to a 1. Although this is not an effective way to uniquely store the original 160 bits, it has several advantages, especially for digital forensics. If you assemble a list of file hashes for a million stolen documents and store them all in a single 4 MiB Bloom filter, only six million bits (at max) in that Bloom filter will be set. Not only will the Bloom filter be much smaller than the list of a million hashes (which would take up 20MiB, instead of 4MiB) and is much more compressible, it's also significantly faster to search. Of course, when searching a Bloom filter there is always the risk of a false positive—some other document might have a hash that, when chopped into four parts, just happens to match four other partial hashes. This kind of false positive can be an advantage, though, if the files that you are hashing are highly confidential: if the criminal who stole some of your confidential documents manages to steal your Bloom filter, that person won't be able to reverse engineer the Bloom filter and have it spill the hashes of all the documents that you consider sensitive. In either event, the Bloom filter's false positive rate can be tuned as needed for the specific application.

My colleague Vassil Roussev spent several years working with hashes and Bloom filters and developed a metric for determining how similar two files are. The metric works by scanning files for what Roussev called "statistically improbable features" and then hashing a window of 64-bytes and storing the hash in a Bloom filter. When a certain fraction of bits in the Bloom filter fill up, Roussev's algorithm starts on the next filter. With this system, the similarity of two files is proportional to the number of bits that are set in common in the filters. One of the neat things about this system is that you can compare Bloom filters for a small file and a very large file and find out if the small file resides *inside* the larger file. This even works if the larger "file" is an image from a multi-terabyte-sized disk array [6].

Roussev's *similarity digest* overcomes a fundamental problem of using cryptographic hashes to find files of interest. By design, if you change just one bit of a file, the file ends up with a completely different cryptographic hash. Such changes can be made intentionally to thwart investigators. The similarity digest doesn't suffer from this problem.

In my own work, I found that a 4 KiB of data extracted from most video files and JPEGs tends to be highly identifying, even possibly unique. So my system chopped sensitive files into 4 KiB chunks, hashed them, and stored the hashes in a high-performance database we built called *hashdb*. You can then search a TB-sized drive to see if it holds any of the videos in your collection by randomly sampling a small fraction of the drive's sectors, hashing them, and looking up the hashes in the database. In theory, this would let us probabilistically search a TB-sized drive for the presence of a sensitive video in just a few minutes [7]. In practice, we found it too difficult to obtain sector hashes of sensitive files due to organizational and administrative issues, so we never deployed this technology.

Digital Timestamping

One use of cryptographic hashes that was pioneered in the 1990s and is coming back into vogue is to use them as the basis of a digital timestamping service.

The roots of using hashes for timestamping go back to 1989, when a researcher at MIT accused Thereza Imanishi-Kari of scientific fraud and misconduct. One of the key allegations was the data in laboratory notebooks had been fraudulently altered. Both the US Congress and the US Department of Health and Human Services (HHS) opened investigations. The US Secret Service raided Dr. Imanishi-Kari's lab and seized her notebooks. Although the HHS Office of Research Integrity (ORI) concluded that fraud had taken place, that finding was overturned on June 21, 1996, by the HHS Research Integrity Adjudications Panel, which found that ORI "did not prove its charges by a preponderance of the evidence" (a relatively low legal standard).

Stuart Haber and Scott Stornetta were cryptographers at Bellcore (Bell Communications Research). They wanted some way that cryptography could protect organizations from the allegations that were flying around MIT of notebook alterations.

For those who have never worked in the physical sciences, let me assure you that physical laboratory notebooks can be serious stuff. Research organizations might distribute individually

serialized notebooks to their scientists, who are expected to date and sign each page. Mistakes are supposed to be crossed out with a single line, so that the erroneous entry can still be read. Corrections must also be dated and signed. One reason for such procedures is to establish clear evidence regarding the date that something that is discovered or invented, which might one day be a key fact in patent litigation. Such procedures are also designed to protect against fraud.

Electronic laboratory notebooks would seem to offer none of the protections of physical notebooks, since digital data can be changed without a trace. One obvious approach is to hash a document with a timestamp and sign the result with a secret key. The problem with this approach is the holder of the secret key—call it the timestamping agency (TSA)—must be trusted not to write a fraudulent signature.

Haber and Stornetta came up with an approach that eliminated the need to trust the timestamping agency. In their first patent (US 5,136,634, filed August 4, 1992), the TSA maintains a special hash called the *catenate value*. When a new document is to be timestamped, the TSA creates a receipt by hashing the document's hash with the current date. The TSA then takes this receipt and hashes it with the previous catenate value to create the next concatenate value. All of the hashes, with all the timestamps, thus make up a hash chain. The system that they ultimately developed, described in US Patent 5,781,629 (filed February 21, 1997), arranges document hashes into a Merkle Tree. The two founded a company called Surety, which is still going strong today.

Satoshi Nakamoto, the pseudonymous author of the original Bitcoin paper [8], references Haber and Stornetta's article [9] as one of Bitcoin's inspirations. For a list of other academic contributions that ended up in Bitcoin, see Narayanan and Clark's article in *ACM Queue* [10].

Finally, here is the puzzle from Stuart Haber:

Let's say it is 2020 and you have a document D with a digital timestamp certificate from 1997. The certificate is based on MD5, a hash function that was secure then but today suffers from known collision attacks, although the algorithm is still pre-image resistant. What do you do? You could certainly timestamp the document today, but that doesn't prove that it was around back in 1997. How do you renew the timestamp in a way that's mathematically defensible?

The solution, says Haber, is to timestamp the concatenation of the 1997 document and its 1997 digital certificate. Because MD5 is still believed to be pre-image resistant, we can't make a document today that has the same MD5 of an arbitrary document from 1997. Timestamping the concatenation today proves to the future that both exist today, and the certificate from 1997 proves today that the document must have existed back in 1997.

References

[1] H. Stevens, "Hans Peter Luhn and the Birth of the Hashing Algorithm," *IEEE Spectrum*, January 30, 2018: https://spectrum.ieee.org/tech-history/silicon-revolution/hans-peter-luhn-and-the-birth-of-the-hashing-algorithm.

[2] R. S. Merkle, "Secrecy, Authentication and Public Key Systems," Technical Report No. 1979-1, Information Systems Laboratory, Stanford Electronics Laboratories, Department of Electric Engineering, Stanford University, June 1979: https://www.merkle.com/papers/Thesis1979.pdf.

[3] W. Diffie and M. Hellman, "New Directions in Cryptography," *IEEE Transactions on Information Theory*, vol. 22, no. 6 (November 1976), pp. 644–654: https://doi.org/10.1109/TIT.1976.1055638.

[4] Google's announcement is at https://security.googleblog.com/2017/02/announcing-first-sha1-collision.html. You can download the two PDFs from https://shattered.it, where you will also find a visualization of the file's internals and links to the program that produced the files.

[5] B. H. Bloom, "Space/Time Trade-offs in Hash Coding with Allowable Errors," *Communications of the ACM*, vol. 13, no. 7 (July 1970): pp. 422–426: https://doi.org/10.1145/362686.362692.

[6] V. Roussev, "Managing Terabyte-Scale Investigations with Similarity Digests," in G. Peterson and S. Shenoi, eds., *Research Advances in Digital Forensics VIII* (Springer, 2012), pp. 19–34: https://doi.org/10.1007/978-3-642-33962-2_2.

[7] J. Young, K. Foster, S. Garfinkel, and K. Fairbanks, "Distinct Sector Hashes for Target File Detection," *IEEE Computer*, vol. 45, no. 12 (December 2012): pp. 28–35.

[8] S. Nakamoto, "Bitcoin: A Peer-to-Peer Electronic Cash System," 2008.

[9] S. Haber and W. S. Stornetta, "How to Time-stamp a Digital Document," *Journal of Cryptology*, vol. 3, no. 2 (1991), pp. 99–111.

[10] A. Narayanan and J. Clark, "Bitcoin's Academic Pedigree," *Communications of the ACM*, vol. 60, no. 12 (November 2017), pp. 36–45: https://doi.org/10.1145/3132259.