An Introduction To Digest Algorithms

Ross N. Williams
ross@rocksoft.com
Rocksoft

12 September 1994

Abstract

One of the most useful tools produced by the field of cryptology is the cryptographic hash, or message digest. Digests are like checksums, but provide such a good fingerprint of the data that it is actually computationally infeasible to change the data without also changing the data's digest. This special property provides new guarantees of integrity that lead to some surprising applications. Many digest algorithms are simple to use, patent free, and have C implementations that are freely available on the Internet by FTP. This paper provides an introduction to digest algorithms, and reviews their application to disk monitoring, intruder and virus detection, file transfer verification, notarization, and authentication. It also gives enough practical details to enable the reader to deploy the popular MD5 digest algorithm.

1 Introduction

In the last twenty years, the field of cryptology has undergone a revolution, sparked by the invention of public key cryptography[**Diffie76**]. This revolution has yielded a range of new techniques that can be applied in various combinations to solve almost any secrecy, integrity, or authentication problem. Some of these techniques are complex, requiring a high degree of cryptographic sophistication to be applied securely. Others are simple, general-purpose tools that can be applied effectively by anyone who takes an interest in them. One such tool is the digest algorithm, which reads a block of data of any size and generates a small (e.g. 16-byte) fixed-width, non-invertible "digest". This digest has particular special properties that enable it to act as a form of identity of the original block. The best way to gain an understanding of digests is to approach them through their ancestors, the checksums.

2 Checksums

Checksum algorithms are a particular class of hashing algorithm that were devised to solve the problem of detecting errors in messages transmitted through noisy communication lines. To enable errors to be detected, the transmitter calculates the checksum of the message and transmits it after the message. To check the message, the receiver calculates the checksum of the received message and compares it with the received checksum. For example, if the checksum algorithm simply summed the bytes in the message **mod** 256, then the transfer might proceed as follows:

```
Msg          :  6 23  4
Msg,checksum :  6 23  4 33
Msg received :  6 27  4 33
```

In this example, the second byte of the message has been corrupted from 23 to 27 by the communications channel. However, the receiver can detect that something is wrong by comparing the transmitted checksum (33) with the calculated checksum of 37 (6 + 27 + 4).

If the checksum itself is corrupted, a correctly transmitted message might be incorrectly identified as a corrupted one. However, such safe-side failures don't matter much. A dangerous-side failure occurs when the message and/or

checksum is corrupted in a way that results in a transmission that is internally consistent. Unfortunately, this possibility is completely unavoidable, and the best that can be done is to minimize its probability by strengthening the checksum algorithm until the chance of a dangerous-side failure is acceptably low.

One way to strengthen the checksum algorithm is to change from an 8-bit register to a 16-bit register (i.e. sum the bytes **mod** 65536 instead of **mod** 256) so as to apparently reduce the probability of failure from 1/256 to 1/65536. While essentially a good idea, widening the checksum alone is not sufficient to significantly increase the strength of the error detection, as the formula used may not be sufficiently random. With a simple summing formula, each incoming byte affects roughly one byte of the summing register no matter how wide the summing register is. In the example above, the error would still go undetected even if the summing register was one Megabyte wide. This problem can only be solved by replacing the simple summing formula with a more sophisticated calculation that enables each incoming byte to have the potential to affect the entire checksum register. Thus, there are actually two requirements for a strong checksum algorithm:

> **Width:** The algorithm should have a register wide enough to provide a low a-priori probability of failure (e.g. 32-bits gives a $1/2^{32}$ chance of failure).
>
> **Randomness:** The algorithm should give each input byte the potential to affect many bits of the register.

In other words, there is a need for enough width in the checksum to provide an acceptably low chance of a-priori dangerous-side failure, and enough randomness to ensure that every byte of the message will have a significant effect on the result. These goals are reflected in contemporary checksum algorithms such as CRC-32. Unlike their predecessors, modern check*sum* algorithms no longer simply *sum* the bytes; in fact, they are more likely to divide them! Despite this, the term *checksum* is still used to describe any algorithm that produces an error correction value, and which does not have the cryptographic strength of the digest algorithms described in the next section. Some common checksums are: CRC-32, CRC-16, Fletcher, and IP.

3 Problems With Checksums

Checksum algorithms provide a good solution to the problem of detecting errors in data introduced by random phenomena such as noise on communication lines or defects in a hard-disk surface. However, they are totally inadequate in the face of hostile humans.

Consider the example of a military commander who despatches the message "`LAND`" by courier to the other half of his army, to indicate that an attack is to take place by land. Being fearful that the courier could be intercepted and the message substituted, the commander takes the precaution of authenticating the message by calculating its checksum (`31`) and transmitting the checksum by smoke signal, a slow, but completely uncorruptable channel. In doing this, the commander relies on the checksum algorithm as an authenticator.

```
Msg in ASCII      :  L   A   N   D
Msg in decimal    : 76  65  78  68
Msg,checksum      : 76  65  78  68  31
Corrupted (Dec)   : 83 101  97  25  31
Corrupted (ASCII) :  S   e   a  ??
```

Unfortunately, the enemy *does* intercept the message, and decides to replace it with a different one: "`Sea`". Being aware of the checksum smoke signal, and realizing that a naive substitution would be detected, the enemy *modifies the message in such a way that it has the same checksum*. It does this by appending the control character `25` to the end of the message to make the bytes of the message sum to 31 (**mod** 256). As a result, the other half of the commander's army receives the message, checks its checksum against the smoke signal checksum, and accepts the substituted message.

This example demonstrates that, whereas checksums provide excellent protection against

random errors, they provide very little protection against intelligent and malicious agents who are not constrained to corrupt messages in a random manner. To protect against intelligent opponents, signature algorithms must be strengthened to the point where, it is *extremely difficult to find or construct a message having a particular signature.*

3.1 Digest Algorithms

To provide greater security, checksum algorithms can be strengthened either by increasing their width or increasing their randomness. Digest algorithms do both, providing extra width and randomness, and raising the level of security to a point of practical unforgeability. This makes them slower than checksum algorithms — the price of the extra security.

Whereas most checksum algorithms are invertible and generate a 2-byte or 4-byte checksum, most digest algorithms generate a 16-byte (128-bit) digest, and are non-invertible. If the digest algorithm is designed properly, it will possess the following two properties:

- It is computationally infeasible to find any message that corresponds to a given digest.
- It is computationally infeasible to find any two messages that correspond to the same digest.

Such algorithms are referred to as **strong one-way hash functions**, which we may equate with the term **digest**. Algorithms that possess only the first of the two properties are referred to as **weak one-way hash functions** and are best ignored by the practitioner. Strong one-way hash functions are sometimes also referred to as **collision-free hash functions**. This is somewhat of a misnomer, as any function that maps a space containing an infinite number of values (e.g. the set of all finite blocks of bytes) to a space containing a finite number of values (e.g. the set of all 128-bit blocks), *must* have an *infinite* number of collisions! In practice, however, you're unlikely to ever actually encounter a colliding pair. You'd have to examine about eight billion values every second to have a greater than 50% chance of finding a collision in your lifetime.[1]

Following a crushing defeat on the battlefield, we find that our commander has mended his ways and is using a digest algorithm instead of the checksum algorithm:

```
Msg in ASCII     :  L   A   N   D
Msg in decimal   : 76  65  78  68
Msg + hex digest : 76  65  78  68
F87C04A7D828D9B050773B33DE2382FE
```

While our commander might find transmitting `F87C04A7D828D9B050773B33DE2382FE` by smoke signal to be a little inconvenient, it's extremely unlikely that the enemy will be able to find a substitute message that has the same digest. As a consequence, any observer who sees the smoke signal digest can have confidence in any corresponding message. The digest acts as a secure form of authentication.

In addition, digest algorithms are constructed so as to make it extremely difficult to recover any part of the original message from the digest. Thus, even if the message is secret, the general can transmit its digest publicly (by smoke signal) with confidence. The only threat to secrecy arises if the number of messages is so small that the opponent could predict some of them, calculate their digests, and hence "recognise" particular messages by their digests. This threat can be eliminated by starting each message with a fixed amount of noise.

4 The Essential Property

From a *practical* point of view, the essential property of digests is:

> **Property:** *From a practical perspective, digest algorithms implement the apparent miracle of providing a one-way one-to-one mapping between the infinite set of data blocks and the finite set of n-bit digests.*

[1] Calculation: A birthday attack on 128-bit space over seventy years would require the evaluation of $\sqrt{2^{128}}/(70*365*24*60*60)$ values per second.

Of course, the mapping isn't really one-to-one, but for all practical purposes we can consider it to be. The "proof" of this is that it's *practically* impossible to demonstrate that the mapping is not one-to-one, as it's *practically* impossible to find a collision!

Thus one way to gain a practical mastery in the application of digests is simply to imagine that they really are one-to-one (even though they're not). For all practical purposes, the digest of a block of data can act as that block's unique *identity*. The various applications of digest algorithms flow naturally from this perspective.

5 Applications

The special properties of digests lead naturally to a variety of applications:

5.1 Disk monitoring

Digests can be used to detect changes in file systems. If the digest of a file is recorded, any future change in the file can be detected by calculating the digest of the file and comparing it to the recorded digest. At first glance, it might seem that checksums would work just as well. However, digests provide two major advantages. First, digests are much wider than checksums, providing far greater assurance that no change has occurred when the values do match. Second, digests provide more security against attacks by intruders and viruses. A clever intruder might find it easy to modify a file in such a way that the modified file has the same checksum as the old file, but would find it practically impossible to fool a digest. The public security tool *Tripwire*[**Kim94**] [**Spafford92**] and the commercial data integrity tool *Veracity*[**Williams94**] [**Rocksoft94**] both record digests to monitor files.

5.2 File transfers

Digests do not have any special application to the verification of file transfers. However, as they are wider than most checksums, digests can provide a far higher degree of confidence in the transfer than ordinary checksums. As the volume of packets transmitted through data networks increases, existing checksums may not provide a sufficiently high degree of checking. For example, a 16-bit checksum with a one in 65536 chance of a false positive would have about an 8% chance of failing to detect at least one error in the transmission of 3000 blocks down a phone line requiring an average of three attempts to transfer each block.[2] For a 32 bit checksum, there would be about an 0.0001% chance.[3] Digests are usually 128-bits or wider and effectively eliminate this uncertainty.

5.3 Notarization

Digests can be used to timestamp (notarize) blocks of data (e.g. documents). To notarize a document, publish its digest in a secure archival medium, such as the classified section of a widely circulated and archived newspaper. Once this is done, the document can be shown to have existed at that time by producing a machine readable copy of the document, along with a reference to the archive. This evidence, combined with the computational infeasibility of creating a document that would match the digest, provides cryptographic proof that the file existed on or before the date when the newspaper was published. Publication of a document's digest is equivalent to timestamping the document.

The beauty of this form of timestamping is that it can be performed silently (i.e. without anyone sighting the document or even noticing) and securely (i.e. the cryptographic "proof" is very strong). Some specific applications of this notarization technique are: 1) stamping the date of invention for patents without disclosing the invention to anyone, 2) stamping critical corporate or government documents so that they can't be forged or tampered with at a later date.

Warning: *Although timestamping using digests provides extremely strong cryptographic proof of the existence of a document at a particular time, and would probably convince most cryptologists, to our knowledge such proof has not yet been tested in court.*

[2] Calculation: $1 - (65535/65536)^{6000}$.
[3] Calculation: $1 - ((2^{32} - 1)/2^{32})^{6000}$.

5.4 Private key authentication

Digests can be used to implement the authentication of messages between two parties sharing a secret key K. To send a message, the transmitter appends the secret key to the message M, computes the digest of the result X=$d(MK)$, and then transmits MX.[4] The receiver receives $M'X'$ and authenticates M' by testing for $X' = d(M'K)$. This scheme works because any active eavesdropper would have only M and X to work with, and would have to find some NY such that $Y = d(NK)$ without knowing K. This is equivalent to the problem of breaking the digest.

One attack on this scheme is the playback attack. An opponent could record one of the messages and play it back later, to some detrimental effect. This attack can be prevented by including some unique stamp in each message that identifies it as being current. Some stamping ideas are: the date and time, a serial number, or one of a set of agreed numbers (with a record being kept of the ones used).

5.5 Public key authentication

Digests can also be used as a component of public key authentication. The most straightforward way to transmit a message with public key authentication is for the transmitter to append some redundant information R to the message M, encrypt MR using the private key, and transmit the result. The receiver can authenticate the transmission by decrypting it using the public key and checking that the received redundant information is consistent with the received message. The disadvantage with this scheme is that some public key methods (e.g. RSA) are quite slow, and, if there is no need for secrecy, it's inefficient to have to encrypt the entire message just to provide authentication. A more efficient alternative is to encrypt just the digest of the message, using the private key. The encrypted digest can be appended to the unencrypted message to provide efficient 16-byte authentication.

[4] Note: In this paper XY usually denotes concatenation, not multiplication.

5.6 Storage of passwords

Multi-user computer systems have to store a database of passwords to enable them to authenticate users. However, it's insecure to keep a file of passwords anywhere in the computer system, as it's possible that an intruder might somehow read parts of the file without first breaking system security (e.g. the intruder might find part of the file in a recycled block on the disk). The solution is to store not the passwords themselves, but their digests. This enables the operating system to check passwords of users logging in, but does not enable an intruder who stumbles across the digests from reconstructing the passwords themselves. This scheme is currently in use in most operating systems (see **[Evans74]** for an early paper on this topic). One attack on this scheme is to feed each word in a dictionary through the digest algorithm and compare the resulting digests with the digests in the password file. This attack can be slowed by generating a random salt for each password entry, storing it with the entry, and calculating digests from the password and the salt. This forces the opponent to reprocess the entire dictionary for each password entry.

5.7 Proof of existence

Digests can act as a proof of the existence of a particular block of data, making them suitable for remote server verification. Using digests, an entity B can prove to an entity A that it possesses a particular block of data (that A also possesses). The most obvious way to do this (short of having B actually transmit the block to A) would be simply to have B send A the digest of the block. Unfortunately, this is not convincing as A might suspect that B has thrown away the data and is merely storing the digest! Stronger proof can be achieved by arranging for A to challenge B by sending B a random number R. To respond, B calculates and returns the digest of RD. A can then perform the same calculation to verify that B still possesses the data. It's important that RD is used rather than DR as if DR were used, B could simply process D once, throw D away, and store only the 128-bit *state* of the digest algorithm.

This technique can be applied to audit backup

services. The client could generate a large backup file and then calculate and record the digests of the file appended to a hundred or so different random values. These random number/digest pairs could then be used on a regular basis to challenge the backup service to prove that it still has the file, even if the client doesn't. Such proof would be secure, efficient and would require just a few bytes of network traffic. To the author's knowledge, this is an original application for digest algorithms.

5.8 Protocol protection

The capacity of digests to act as proof of the existence of data also leads to a protocol protection scheme. Suppose that you are developing a commercial product of some kind that uses some kind of protocol or file format, and you don't want competitors producing interoperable products. One way to do this is to patent an algorithm and then build it into the protocol. For example, if the protocol relies on compression, a data compression patent could be used to stop interoperable products. However, this is a rather "overengineered" approach. Digests provide the possibility of a simpler way.

To use a digest to protect a protocol, make the calculation of the digest of blocks of data and a key K an essential part of the protocol. For example, K could be appended to each network packet and the digest of the result transmitted with each packet. The next step is to choose a K that is copyrighted! K could be part of your program, a technical paper, a copyright message, or anything else that is obviously a copyrighted work and is small enough to be embedded in executable programs. Once all this is in place, the creation of an interoperable product will be either computationally or legally infeasible, for it will be computationally infeasible to interoperate without K, but if K is used, you can sue them for copyright infringement of K! Even if an attempt is made to hide K in the compatible product, the non-invertibility of digests, and their use as a proof of existence of information could be used to prove that the compatible product must contain K somewhere! To the author's knowledge, this is an original application for digest algorithms and has never been tested in court.

5.9 Data structures

Because it's extremely improbable that collisions (different inputs, same output) will ever be encountered in practice, digest algorithms can be used in data structures to generate *unique* fixed-length identifiers for arbitrary blocks of data in situations where the identifier of identical blocks must be the same, but arbitrary identifiers (e.g. serial numbers) cannot be assigned because there is no central entity to issue identifiers.

For example, a network of computers implementing a distributed database of documents could collect documents from their various input sources during the day and then synchronize at night by exchanging and comparing the digests of the new documents. Checksums could be used for this purpose, but would require a follow-up comparison between documents whose checksums matched. Digests provide enough assurance of uniqueness to eliminate the need for any such follow-up comparison.

This example of a document synchronization system is just one application for digests in data structures, and it's likely that they could be applied in many other data structure applications as well. The essence is that the digest can act as a convenient identity for a block of data.

5.10 Summary

Perhaps the most exciting aspect of these applications is that they do not require a great degree of cryptographic sophistication to implement. All that one needs is a digest algorithm and an understanding of the properties of digests mentioned earlier.

6 How Secure Are Digest Algorithms?

In assessing the security of digest algorithms, we must consider their strength in the face of random and intelligent attacks.

The security of digest algorithms against random attacks can be measured in the same way

as for checksums. As most digest algorithms produce 128-bit (16-byte) digests, the chance of a random error in the data going undetected is astronomically small, being just one in 2^{128} (about one in 3.4×10^{38}). To be this unlucky, you would have to win a million-ticket lottery six times in a row![5] Thus, because of their increased width, digests generally provide far more protection against random attacks than checksums.

Assessing the strength of a digest algorithm against attacks by an intelligent opponent is very much harder, as it is possible that any digest algorithm could have a hidden weakness that could be exploited. Currently, the only way to determine if a digest algorithm is secure is to expose it to public scrutiny over a period of years. Unfortunately, the field of digests is relatively young and so most digests are only a few years old.

If we ignore the potential for hidden weaknesses, and assume that the digest algorithm being assessed provides perfect randomness, we need also to take a look at that other determinant of security: width. High randomness in a digest algorithm is no use if the opponent can use brute force to search the digest space. As an n-bit digest can take 2^n different values, any opponent attempting to search this space will find a match after an average of 2^{n-1} attempts. For a 128-bit digest this is 2^{127} (1.7×10^{38}) attempts. This is highly secure. Even if an opponent could test 10^9 new messages per second (which in 1994 technology would require at least one thousand high speed CPUs), it would still take about 5.4×10^{21} years.[6]

Unfortunately, there is a different kind of brute-force attack, called a "birthday attack" that reduces the amount of work required from $O(2^n)$ to $O(\sqrt{2^n})$. Birthday attacks are named after the famous birthday paradox which states that a group of just 23 people are needed for there to be a probability of at least one half that at least two people in the group share the same birthday. It's considered a paradox because common sense suggests that many more people would be required. Birthday attacks do not provide an opponent with a faster-than-brute-force way of finding a document having a *particular* digest, but they do allow an opponent to *create* a *pair of documents* having the same digest in about $\sqrt{2^n}$ attempts. To execute a birthday attack, the opponent generates $\sqrt{2^n}$ random documents and computes each document's digest. The set of such digests is then searched for duplicates. A result related to the birthday paradox states that the probability that there will be at least one matching pair will be about 0.5.

In a variation of this attack, two documents can be produced that are approximations of two target documents, and which have the same digest. This would enable an opponent to present one of the two documents officially and then later substitute the second document **[Yuval79]**.

How exposed to birthday attacks are existing digest algorithms? As it happens, not very. But birthday attacks do bring digest algorithms much closer to the brink. If a digest algorithm produces a 128-bit digest, then a birthday attack will require an average of 2^{64} operations to succeed (instead of 2^{128-1}). At 10^9 tests per second, 2^{64} tests would take over 500 years.[7] Thus, a very determined opponent with a network of one thousand workstations, with a solid year of CPU time to burn, would have a one in 500 chance of finding two documents with the same digest. Whether you consider this to be a serious risk depends on the criticality of your data.

For further information on attacks on digest algorithms, see **[Pieprzyk93]**. For more information on how digest algorithms fit into the field of cryptology, see **[Simmons92]**.

7 Technical Restrictions

This section contains a brief discussion of technical restrictions on the use of digest algorithms.

Memory consumption: Most digest algorithms require very little memory, using it only for their internal state (e.g. 128-bits), and to buffer incoming bytes into blocks (e.g. 64 bytes).

CPU consumption: Digest algorithms are much slower than checksum algorithms as digest algorithms have to provide a computational

[5] Calculation: $10^{6^6} = 10^{36} \approx 10^{38}$.

[6] Calculation: $(2^{127}/10^9)/(365 * 24 * 60 * 60) \approx 5.4 \times 10^{21}$.

[7] Calculation: $(2^{64}/10^9)/(365 * 24 * 60 * 60) \approx 580$.

barrier for their opponent rather than merely a probabilistic one. However, digests are still fast enough to warrant the routine replacement of checksums by digests in many applications. The design of secure, high-speed digest algorithms is an active research area **[Anderson94] [Pieprzyk93] [Sci.crypt]**.

8 Legal Restrictions

Copyright: Many of the more popular digest algorithms have implementations whose C source code is available through the Internet by FTP. Many of these implementations come with public copyright licences that apply only minor restrictions to the programmer, such as requiring that the algorithm be properly identified. The MD5 algorithm is one such algorithm and its copyright message is reproduced here in full as an example:

> Copyright (C) 1991-2, RSA Data Security, Inc. Created 1991. All rights reserved.
>
> License to copy and use this software is granted provided that it is identified as the "RSA Data Security, Inc. MD5 Message-Digest Algorithm" in all material mentioning or referencing this software or this function.
>
> License is also granted to make and use derivative works provided that such works are identified as "derived from the RSA Data Security, Inc. MD5 Message-Digest Algorithm" in all material mentioning or referencing the derived work.
>
> RSA Data Security, Inc. makes no representations concerning either the merchantability of this software or the suitability of this software for any particular purpose. It is provided "as is" without express or implied warranty of any kind.
>
> These notices must be retained in any copies of any part of this documentation and/or software.

An important aspect of this notice is that it does not restrict the commercial use of MD5. This means that you can incorporate MD5 into commercial products so long as you conform with the minimal requirements above.

Patents: Software patents provide far greater restrictions than copyright as they protect the algorithm itself, rather than a particular implementation. Although quite a few digest algorithms are patented (e.g. MDC-2 and MDC-4 (by IBM)), these patents are usually fairly narrow, and the field seems to be remarkably free from the kind of broad software patents that have muddied fields such as data compression. In particular, many digest algorithms (e.g. MD5 and SHA-1) appear to be entirely patent free.

Exportability: The US Government's current policy is to restrict the export of cryptographic products for defence reasons. This policy has been propagated through treaties to associated countries such as Australia and has caused uncertainty about what software can and can't be exported.

Although digest algorithms embody cryptographic strengths, it seems that there is no problem with exporting them, so long as they do not provide secrecy, just authentication. However, it would be wise to check with the relevant authorities anyway.

9 A Short Catalogue of Digest Algorithms

This section contains a description of some of the more common digest algorithms.

MD2: This is the "RSA Data Security, Inc. MD2 Message-Digest Algorithm". The defining document for MD2 is RFC-1319**[Kalisky92]**. MD2 is secure, but slow. Its width is 16 bytes (128 bits).

MD4: This is the "RSA Data Security, Inc. MD4 Message-Digest Algorithm". MD4 is a fast, fairly secure digest algorithm. However, when it was released, it became so popular that its inventors became concerned that it might not be secure enough for widespread use and released a more secure version which they named MD5. As such, MD5 is preferred. The defining document for MD4 is RFC-1320**[Rivest92a]**. Its width is 16 bytes (128 bits).

MD5: This is the "RSA Data Security, Inc. MD5 Message-Digest Algorithm". This is a fast and secure algorithm. The defining document for MD5 is the RFC-1321**[Rivest92b]**. Its width is 16 bytes (128 bits).

SHA, SHA-0, SHA-1: (Secure Hash Algorithm). This is a digest algorithm proposed by the US NIST (United States National Institute of Standards and Technology) agency as a standard digest algorithm. The terminology here is a little confusing. NIST released a first version of SHA which it referred to as "SHA" **[NIST93]** . It subsequently found a problem with it and released a second version which it called "SHA-1" **[NIST94]**. As it's likely to be confusing to talk about "SHA" and "SHA-1", it makes sense to call the earlier algorithm "SHA-0" and use the term "SHA" only to refer to the entire group of digest algorithms.

SNEFRU: This is the Snefru algorithm, also known as "The Xerox Secure Hash Function". Although the name "Snefru" looks as if it is an acronym, it is actually the name of a Pharaoh of ancient Egypt. The Snefru algorithm is described in detail in **[Merkle??]**. Unlike the other algorithms, Snefru has two parameters called the *security level* (number of transformation iterations) and the *output size* (width of the digest in longwords (4-byte chunks)), each of which can take either the value 4 or 8. This leads to four different versions of Snefru. Snefru is one of the few digest algorithms that yield a digest longer than 16 bytes. This in itself could provide extra security, particularly if you are concerned about birthday attacks.

Other Digest Algorithms: Other digest algorithms that the author has heard of, but not yet tracked down are: 3-WAY, A5, BLOWFISH, FEAL, FISH, Haval, MDC-2, MDC-4, N-hash, Purdy Hash, RIPE-MAC1, RIPE-MAC3, RIPE-MD, RSADSI, SAFER, SEAL, VINO, and WAKE. There are apparently many more. Some of these are reputed to be better (faster, more secure) than the digests described earlier. Some are them are reputed to have been broken.

10 Getting Started

The best way to get started with digest algorithms is simply to try one out. A good algorithm to start with is the RSA Data Security, Inc. MD5 Message-Digest Algorithm ("MD5" for short). It is a suitable "default" algorithm because it is simple, well-defined, well-known, patent-free, not too slow, has been exposed to the cryptological community for a few years, and has source code readily available by FTP through the Internet. The MD5 algorithm is described in RFC-1321, which also contains an implementation in C and a small test suite so that you can confirm that you've got it working correctly. RFC-1321 can be obtained by FTP from: `munnari.oz.au:/rfc/rfc1321.Z`.

11 Conclusions

Digest algorithms are one of the simplest and most flexible tools produced by the revolution in cryptography. By providing small, fixed-width, computationally secure identities for the infinite set of data values, digest algorithms provide a tractable identity that can be used in a variety of applications including disk monitoring, intruder and virus detection, file transfer verification, notarization, authentication, password management, and data structures. All this power is delivered by simple, easy-to-use algorithms whose patent-free implementations in C are freely available on the Internet by FTP. The simplicity, power, and availability of digest algorithms means that they should now be a standard tool in every programmer's toolkit.

12 Glossary

Birthday attack: A form of attack on digest algorithms in which randomly generated inputs are fed through the digest algorithm to generate a set of input/output pairs whose output components are then searched for duplicates. Birthday attacks enable digest algorithms that are n bits wide and which have an a-priori difficulty of $O(2^n)$ to be cracked with just $O(\sqrt{2^n})$ effort. That's a great birthday present for any cryptanalyst.

Broken: A digest algorithm is said to be broken when someone has publicised a computationally feasible method for finding two inputs

that map to the same output. A digest algorithm is sometimes also considered to be broken if two such inputs are ever shown to exist.

Checksum: The number produced by a checksum algorithm. Checksums are typically 1, 2, or 4 bytes wide.

Checksum algorithm: A hash algorithm that does not attempt to provide cryptographic protection against inversion. The term "checksum" originally referred to checking algorithms that summed the bytes, but is now often used to refer to any non-cryptographic checking algorithm.

CRC: (Cyclic Redundancy Code) A class of "checksum" algorithms that operate by treating the message as a large binary number and then dividing it in binary without overflow by a fixed constant. The remainder is the "checksum".

Digest: The number produced by a digest algorithm. Typically this number is 16 to 32 bytes wide.

Digest algorithm: A digest algorithm is a hash function that is computationally infeasible to invert. That is, given a digest produced by a digest algorithm, it is very difficult (i.e. computationally infeasible) to find a message that has that digest.

Hash function: In general Computer Science, this term is used to refer to a non-cryptographic function that is used to locate elements in a so-called "hash table". However, in the field of cryptography, it has a much stronger cryptographic flavour and is usually used to refer to a digest algorithm. Even so, those using the term often still feel the need to to add a qualifier such as "strong" or "secure" as in "The Xerox Secure Hash Function".

MAC: Message Authentication Code. A digest that has been calculated from a block of data and a secret key, enabling any receiver possessing the key to authenticate the message.

MD5: An abbreviation for the "RSA Data Security, Inc. MD5 Message-Digest Algorithm".

MDC (Manipulation Detection Code): Another term for "digest". This acronym is favoured by IBM who have named their digest algorithms MDC-1, MDC-2, and MDC-4.

Message: An abstract term used in the cryptographic literature to denote the data fed into a checksum or digest algorithm.

Message digest: Another term for "digest".

SHA: (Secure Hash Algorithm). This is a digest algorithm proposed by the US NIST (National Institute of Standards and Technology) agency as a standard digest algorithm. The terminology here is a little confusing. NIST released a first version of SHA which it referred to as "SHA" **[NIST93]**. It subsequently found a problem with it and released a second version which it called "SHA-1" **[NIST94]**. As it's likely to be confusing to talk about "SHA" and "SHA-1", it makes sense to call the earlier algorithm "SHA-0" and use the term "SHA" only to refer to the entire group of digest algorithms.

SHS (Secure Hash Standard): A NIST standard that specifies the Secure Hash Algorithm (SHA).

Signature: The term "signature" is generally used in the field of cryptography to describe a method in which an agent can digitally "sign" a document in a manner that does not allow him to deny the signature at a later date. However, in the context of integrity checking tools, the term is used more loosely to denote the set of checksum and digest algorithms.

Snefru: Snefru is a digest algorithm produced by Xerox. It's official name is "The Xerox Secure Hash Function".

Tripwire: Tripwire is a free public data integrity program that employs digest algorithms to detect changes in Unix file systems**[Kim94]** **[Spafford92]**.

Veracity: Veracity is a commercial data integrity software tool that was created by the author of this paper. Veracity employs digest algorithms to detect changes in file systems. For more information, see **[Williams94]** or **[Rocksoft94]** or contact the author or Rocksoft.

Width: The width of a checksum or digest algorithm is the number of bits that the algorithm produces as its signature.

13 References

[Anderson94] Anderson R. (editor) "Fast Software Encryption", Proceedings of the Cam-

bridge Security Workshop, Cambridge U.K., 9-11 December 1993, Lecture Notes in Computer Science Number 809, Springer Verlag, 1994.

[Diffie76] Diffie W., Hellman M.E., "New Directions in Cryptography", *IEEE Transactions on Information Theory*, IT-22(6), pp. 644–654, Nov 1976.

[Evans74] Evans Jr. A., Kantrowitz W., Weiss E., "A User Authentication Scheme Not Requiring Secrecy in the Computer", *Communications of the ACM*, Volume 17, Number 8, August 1974.

[Kalisky92] Kalisky B., "The MD2 Message-Digest Algorithm", Internet RFC-1319, April 1992. Available by anonymous FTP from `munnari.oz.au:/rfc/rfc1319.Z`.

[Kim94] Kim G.H., Spafford E.H., "The Design And Implementation Of Tripwire: A File System Integrity Checker", 29 August 1994. Available by anonymous FTP from `coast.cs.purdue.edu /pub/COAST/Tripwire/Tripwire.ps.Z`.

[Merkle??] Merkle R.C., "Snefru 2.5: The Xerox Secure Hash Function", available by anonymous FTP from `parcftp.xerox.com` in `/pub/hash` (it's not clear what the year is).

[NIST93] Federal Information Processing Standards Publication (FIPS PUB) 180: Secure Hash Standard, Computer Systems Laboratory, US National Institute of Standards and Technology, 11 May 1993.

[NIST94] Federal Information Processing Standards Publication (FIPS PUB) 180-1: Secure Hash Standard (Draft), Computer Systems Laboratory, US National Institute of Standards and Technology, 26 May 1994.

[Pieprzyk93] Pieprzyk J, Sadeghiyan B., "Design of Hashing Algorithms", *Lecture Notes In Computer Science*, Number 756, Springer Verlag, 1993.

[Rivest92a] Rivest R., "The MD4 Message-Digest Algorithm", Internet RFC-1320, April 1992. Available by anonymous FTP from `munnari.oz.au:/rfc/rfc1320.Z`.

[Rivest92b] Rivest R., "The MD5 Message-Digest Algorithm", Internet RFC-1321, April 1992. Available by anonymous FTP from `munnari.oz.au:/rfc/rfc1321.Z`.

[Rocksoft94] Rocksoft Pty Ltd, "Veracity User Manual", Rocksoft Pty Ltd, 16 Lerwick Avenue, Hazelwood Park 5066, Australia (`rocksoft@rocksoft.com`).

[Sci.crypt] `sci.crypt` is an internet newsgroup on cryptology which often includes discussions of digest algorithms.

[Simmons92] Simmons G.J. (editor), "Contemporary Cryptology: The Science of Information Integrity", IEEE Press, 445 Hoes Lane, PO Box 1331, Piscataway, NJ 08855-1331, USA, (ISBN: 0-87942-277-7), 1992.

[Spafford92] Spafford G., Kim G., "Tripwire" (a security tool that uses digests and runs on a wide range of Unix machines), available by anonymous FTP from `ftp.cs.purdue.edu` in `/pub/COAST/Tripwire`, 1992.

[Williams93] Williams R.N., "A Painless Guide to CRC Error Detection Algorithms", available by anonymous FTP at `ftp.rocksoft.com` in `/pub/rocksoft/`, 1993.

[Williams94] Williams R.N., "Data Integrity With Veracity", Proceedings of the Digital Equipment Computer Users Society Australia, 1994. This paper is also available by anonymous FTP at `ftp.rocksoft.com` in `/pub/rocksoft/`.

[Yuval79] Yuval G., "How to Swindle Rabin", *Cryptologia*, 3(3), July 1979, pp. 187–190. Note: The author has not yet sighted this paper.

14 Other Information

Version: This is Version 1.0 of this paper (12 September 1994). This paper is available by FTP from the Rocksoft FTP archive. Look in `ftp.rocksoft.com:/pub/rocksoft/`.

Trademarks: *Rocksoft* and *Veracity* are registered trademarks of Rocksoft Pty. Ltd.

Presentation: This paper was presented at 4:00pm on Tuesday 23 August 1994 at the

Australian Decus Symposium 1994 at the Australian National Convention Centre, Canberra, Australia.

Biography: Ross N. Williams obtained his Ph.D. in Computer Science from the University of Adelaide in 1990. His thesis on text data compression was subsequently published as a book by Kluwer Academic Press, Boston. After completing his thesis, he worked as a safety critical software analyst for Australian National and then as an independent consultant to industry and government. He has spent the last two years developing the data integrity product *Veracity* which is now being marketed by his company *Rocksoft*.

Contact information:

Dr Ross N. Williams
Rocksoft Pty Ltd
16 Lerwick Avenue
Hazelwood Park 5066
Australia.

Internet: `ross@rocksoft.com`.
Phone: +61 8 379-9217.
Fax: +61 8 379-4766.