Cracking Hasing Functions

Toby Gilham, Jonathan Poulter and Alan Saul

1 Hashing a message

Hashing functions are able to take a variable length message and from it create a fixed length output. A hashing function will always hash the same input to the same output. The special type of hashing functions, discussed here are cryptographic hashing functions, these have additional properties which mean they are used in a range of security applications.

A hashing function is a mapping for a binary string, H: $\{0,1\}^* \rightarrow \{0,1\}^n$, to create a fixed length digest. The number of distinct possible outputs of a hashing function of fixed length L is 2^L theoretical digests. There are clearly vastly more valid inputs as the input may be of an arbitrary length, therefore there are necessarily collisions, due to the pigeon-hole effect. It is important however that there is no determinable connection between the message, m, and the digest, $H(m)$. H should generate sufficiently random digests such that it fulfils three criteria:

- it should be computationally infeasible to find some m_1 and m_2 where $m_1 \neq m_2$ and $H(m_1) = H(m_2)$, this is the **collision-free property**. This property is desirable because of hashes being used to distinguish input data. Collisions reduce the confidence one can have in a function for its ability to distinguish distinct inputs.
- given a digest h, it should be computationally infeasible to find any m where $h = H(m)$. This is **first pre-image resistance**, this ensures it is difficult to retrieve a hashed message.
- an input m_1, and the digest $H(m_1)$ it should be computationally infeasible to find a second message m_2 where $m_1 \neq m_2$ and $H(m_1) = H(m_2)$, this is **second pre-image resistance**. When a message and hash are found, it should be infeasible to find more messages which hash to the same value.

A tasking being computationally infeasible refers to an adversary being unable to

perform the action in a reasonable amount of time with reasonable resources.

The Random Oracle An ideal fixed-length hashing function will play the role of a Random Oracle which acts as a theoretically optimal cryptographic primitive. A Random Oracle, for a given input, produces a truly random output and the input and corresponding output are recorded. If the same input message is presented to the oracle at a later point the previously generated value is returned, however for each novel input presented a new random corresponding output will be generated.

There are many cryptographic hash functions, many vary in success at fulfilling the properties described. Yet they are integral cryptographic primitives on which many secure protocols are built. Successful cryptographic functions have many uses. In most operating systems user passwords are not stored in plain text, doing so would allow a user who maliciously gained access to the password store to know a user's password. Instead password hashes are stored and on login a user submits the password, which is hashed and verified against the stored hash, if these match then the user is authorised. Hashes are used for the signing digital documents and distinguishing large files without having to do explicit bit-wise comparison between each file, this method provides a method of verifying another's copy of a file and is able to identify matches or inconsistencies.

2 The Merkle-Damgård Construction

Various constructs can build a hashing function, however the most popular over the last two decades, which is the construction underlying the most dominant hashes in use today, is the Merkle-Damgård (MD) construction [8]. The aim is to build a hashing function H which takes an arbitrary length message to create a digest, $\{0,1\}^* \xrightarrow{H} \{0,1\}^n$ in the same fashion as a random oracle would. In practice creating a straight-forward hashing or block cipher to take arbitrary input is impractical, most modern hashing functions, take some compression-function and repeatedly apply it to successive blocks of the message.

If a compression function f is suitably provably random then it is possible to create a equally provably random H by using the MD construction. The function takes a message an splits it into blocks of equal length therefore the construct only works on messages which have length, L where

$$L \equiv 0 \pmod{n}.$$

2.1 MD-Compliant padding

It is therefore necessary to apply padding to the original message. In the initial proposal Merkle simply padded the message with zeros[8]. This is sufficient to keep the important property of the construct, if f is collision resistant, so is H, but more generally MD-compliant padding follows the following three rules. For messages m, m_1 and m_2 in the possible message space.

- m must be a prefix of $\text{Pad}(m)$.
  The padding must be applied to the end of the message.
- **if** $|m_1| = |m_2|$ **then** $|\text{Pad}(m_1)| = |\text{Pad}(m_2)|$.
  Two messages of the same length should pad to equal lengths also.
- **if** $|m_1| \neq |m_2|$ **then** last block of $\text{Pad}(m_1) \neq$ last block of $\text{Pad}(m_1)$.
  This is to prevent a very simple suffix collisions, demonstrated in section 3.1.

2.2 The General Hashing Function

Algorithm 1 shows the basic form of the MD construction. Though many modern hashing algorithms are based on the MD principle, including the SHA series [1] of hashes, these often make changes to the construct to side-step specific vulnerabilities in the original design. Here f can be any collision-resistant compression function which will take an initialising vector and the current message block, b_i and produce a new IV vector for the construct on each iteration of the loop, a collision-resistant hashing function. Figure 1 illustrates the blocking of the original message, the presence of the padding element and the processing of the IVs.

Algorithm 1 The Merkle-Damgård construction

```
function H(b_1, ..., b_l)
    y ← IV_0                          ▷ IV_0 is some initialising vector
    for i = 1 → l do
        y ← f(y, b_i)
    end for
    return y
end function
```

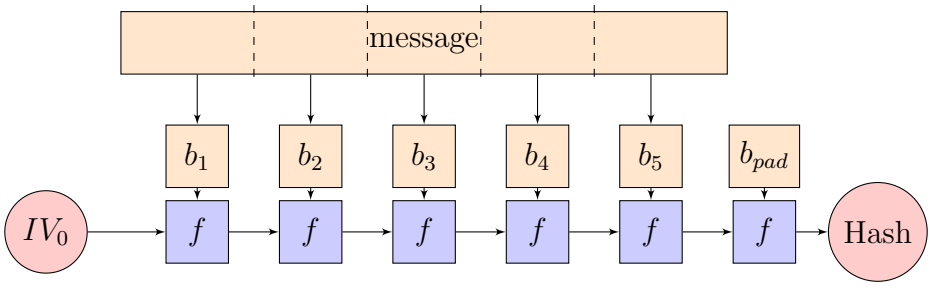

Figure 1: The Merkle-Damgård construction: firstly the message is split into many blocks, each is sequentially processed by a compression function f with some initial IV, IV_0, this produces new IV values, which are passed the the next usage of f. Finally the IVs are finalised using the MD padding block and the resulting IV is the given hash

2.3 Weaknesses of the MD Construction

As cryptanalysis has progressed the number of attacks on the MD construction itself has increased[4][15]. It is important to note that these are not attacks on the specific compression functions used, though these also exist, but these attacks are directly against the MD structure and therefore would reduce the work needed to find collisions even using a perfect random-oracle for the compression function.

2.4 Length-Extension Attack

The first weakness of the simple MD construction is that once a single collision has been found it is trivial to find more[11].

Let $H : \{0,1\}^* \to \{0,1\}^n$ be a hashing function and two distinct messages m_1 and m_2 are a collision pair, that is $H(m_1) = H(m_2)$, under the simple MD construction it is trivial to generate more collisions by appending the collision-causing message with a common message x producing $H(m_1||x)$, $H(m_2||x)$. Where $||$ is the concatenation operator.

3 The MD5 Hashing Function

The MD5 hashing algorithm[10] created in 1992 is an MD based hashing function. It has been widely used across the internet to store password hashes and is still

dominant in providing file checksums, even though it has been shown to be collision weak MD5 is till commonly used in many situations. In the following section the algorithm itself is described as per the specification laid out in RFC 1321 [10]. A clear python implementation is included at the end of this report, the intention is to demonstrate a simple implementation which is useful for education rather than speed of execution.

3.1 MD5 message preparation

The MD5 digest algorithm processes a b-bit original message.

Padding A mandatory "1" bit is added to the end of the bit-string Padding "0" bits are added to the message to make the length, l,

$$l \equiv 448 \pmod{512}.$$

Length appending The message will be processed in blocks of 128-bits, to create a 128-bit output so the message length must be a multiple of 128-bit also. Also it is desirable to include the length-strengthening discussed in 2.4. This has the ability to prevent some attacks, which the MD construction is weak against. So the bit-string is appended with the 64-bit representation of the number b. Now,

$$l \equiv 0 \pmod{512}.$$

IV initialisation Once the message has been prepared the MD construction is initialised, the initial vector of the construct (IV_0) is

$A_0 = \texttt{0x67452301}$ $B_0 = \texttt{0xefcdab89}$ $C_0 = \texttt{0x98badcfe}$ $D_0 = \texttt{0x10325476}$

3.2 The MD5 Compression Function

Once the message has been prepared the MD5 compression function is ran over the padded message, 128-bits at a time. The compression function is illustrated in figure 2 and is initialised to some given some IV, shown by A, B, C, D. A, from the IV is permuted by the function and the result is stored into the location of B. Firstly a non-linear combination of B, C and D is added, then the 32-bit block under consideration in the operation is added. A constant, derived from the mathematical sin function is added. A is then subject to a left-bit rotation, by a

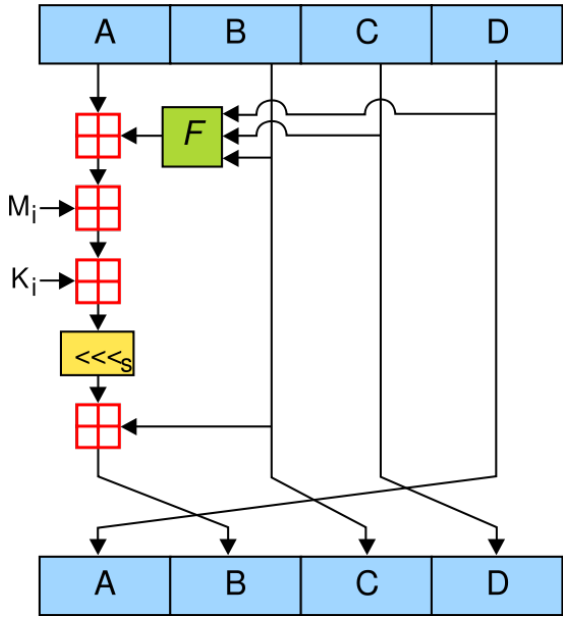

Figure 2: The MD5 compression function.

varying number of places and then the contents of B is also added. 64 variants of this operation are performed, they are grouped into blocks of 16, producing 4 'rounds'. It is important to note that all addition is modulo 2^{32}, 32-bit addition. The resulting hash is the concatenation of the IV components, $A||B||C||D$.

4 Attacking MD5

An attack on a hacking function attempts to circumvent the secure properties that a hashed message should have: the one-way property and the collision-free property, discussed in section 1. A first pre-image attack focuses on finding a message which has a particular hash-value, therefore circumventing the one-way property and tries to find m where $H(m)$ is known. A further attack, a *second-preimage* attack, where the original message m_1 is known, attempts to find another message m_2 where $H(m_1) = H(m_2)$. A collision attack attempts to find any two arbitrary messages m_1 and m_2, where $H(m_1) = H(m_2)$.

IV'_0 :	$A'_0 =$ 0x12AC2375	$B'_0 =$ 0x3B341042	$C'_0 =$ 0x5F62B97C	$D'_0 =$ 0x4BA763ED
m_1:	$X_0 =$ 0xAA1DDa5E	$X_4 =$ 0x1006363E	$X_8 =$ 0x98A1FB19	$X_{12} =$ 0x1326ED65
	$X_1 =$ 0xD97ABFF5	$X_5 =$ 0x7218209D	$X_9 =$ 0x1FAE44B0	$X_{13} =$ 0xD93E0972
	$X_2 =$ 0x55F0E1C1	$X_6 =$ 0xE01C135D	$X_{10} =$ 0x236BB992	$X_{14} =$ 0xD458C868
	$X_3 =$ 0x32774244	$X_7 =$ 0x9DA64D0E	$X_{11} =$ 0x6B7A669B	$X_{15} =$ 0x6B72746A
m_2:	$m1$ with the following change $m_2[X_{14}] = m_1[X_{14}] + 2^9$			
MD5:	0xBF90E670 0x752AF92B 0x9CE4E3E1 0xB12CF8DE			

Figure 3: The collision demonstrated by Dobbertin[4]. This demonstrates a "collision of the compress function" and not on the full hash itself as it relies on getting the algorithm into a point of using these IVs.

4.1 Collision Attacks

Hoffman,of the Network Working Group, in 2004 discussed that the dominant hashing attacks are collision attacks. Considering the 'birthday problem' it is possible to estimate how many evaluations of the hash function it would take to find a collision in a given hashing function. Given a message space of 2^n, where $n = 128$ for MD5, the 'birthday bound' for a binary string is $2^{(n/2)}$; 2^{64} for MD5. This is a theoretical bound on finding a collision in a random oracle. In practice this is reliant on the 'perfection' of the hashing function, that is that each possible 128-bit output should be equally likely and there should be no inferable connection between input and output.

MD5 is not a perfect hashing function and many collisions have now been found, the first publicly announced collision was from Dobbertin in 1996 [4]. The MD5 function processes the message to be digested into 64-byte chunks (blocks) and the collision detailed by Dobbertin relies on using previous blocks to put initialisation vectors of the MD construction into a specific state, from this state their are two distinct blocks which are able to cause a collision. This is not considered to be an attack on the full MD5 function; as a single message isn't presented which causes a collision, but from 1996 the MD5 construction has been widely considered 'broken' with respect to being collision-resistant.

There have been many distributed projects which have tasked themselves with finding MD5 collisions on the full algorithm and the first of these to report results was MD5CRK[14]. This project yielded two full colliding message pairs for the MD5 algorithm, one is shown in figure 3. Collisions were also presented for the MD4, RIPEMD and HAVAL-128 protocols. Each message presented for MD5 was

M_0:	X_0 = 0x6165300E	X_4 = 0x6503CF04	X_8 = 0x2F94CC40	X_{12} = 0x6D658673
	X_1 = 0x87A79A55	X_5 = 0x854F709E	X_9 = 0x15A12DEB	X_{13} = 0xA4341F7D
	X_2 = 0xF7C60BD0	X_6 = 0xFB0FC034	X_{10} = 0x5C15F4A3	X_{14} = 0x8FD75920
	X_3 = 0x34FEBD0B	X_7 = 0x874C9C65	X_{11} = 0x490786BB	X_{15} = 0xEFD19d5A
M_1:	X_0 = 0x6165300E	X_4 = 0x6503CF04	X_8 = 0x2F94CC40	X_{12} = 0x6D658673
	X_1 = 0x87A79A55	X_5 = 0x854F7$\underline{4}$9E	X_9 = 0x15A12DEB	X_{13} = 0xA4341F7D
	X_2 = 0xF7C60BD0	X_6 = 0xFB0FC034	X_{10} = 0x$\underline{D}$C15F4A3	X_{14} = 0x8FD75920
	X_3 = 0x34FEBD0B	X_7 = 0x874C9C65	X_{11} = 0x490786BB	X_{15} = 0xEFD19d5A
MD5:	0xF999C8CA 0xF7939AB6 0x84F3C481 0x1457CB23			

Figure 4: The single-block collision demonstrated by Xie[15]. Modified bits are underlined, this entire collision actually only differs by two bits.
$0 \rightarrow 4$: 0000 $\rightarrow$ 0100 and $5 \rightarrow D$: 1101 $\rightarrow$ 0101.

made of two blocks, each first block was identical and had the effect of forcing the MD construction into using a fixed IV while processing the second block which can be given two distinct blocks and still produce the same hash. The MD5CRK algorithm spent the majority of its time finding a block which is able to generate an exploitable IV, after this is found it took very little time to produce a second block which is able to break collision-resistance.

The same group, 6 years later presented the first public MD5 collision using a single block [15], this collision is shown in figure 4. The implication of being able to generate these collisions easily has already been exploited, most notoriously in the production of fake signed website certificates [12], as this becomes more feasible and computational tractable the chance of generating even more 'useful' collisions increases. A useful collision is not only a collision in the compression function, but a collision which can be used for some form of gain.

4.2 Pre-image attacks

Producing and publishing a collision, may not present a security problem for many uses of hashes, producing a collision for a hashed password may be enough to enter a single system but doesn't necessarily give the attacker the plaintext password. The hashed information is still safe. Once victim to collision attacks most cryptographers will avoid a hash[12], yet more crippling to the reputation of the hashing function is a pre-image attack.

For a given H: $\{0,1\}^* \rightarrow \{0,1\}^n$ a pre-image attack on a hash, h, is able to retrieve m where $h = H(m)$. For an ideal H (which strictly models a Random Oracle) a

brute-force attack is necessary, for MD5 this should take 2^{128} evaluations of the MD5 function for an arbitrary m.

Performing pre-image attacks on arbitrary hashes is clearly a time-consuming task it is therefore desirable to perform some pre-computation to make performing multiple attacks faster, this time-memory tradeoff attempts to save chains of hashes which cover all of a sub-space of the entire message-space.

For a hash H: $\{0,1\}^* \to \{0,1\}^n$ and a finite password set P; one example might be alpha-numeric strings less than 9 characters in length. The perfect data-structure is, for each password in P store p and $H(p)$, this is infeasible and would require storage $\mathrm{O}(|P| \times n)$ bits of storage.

Hash chain tables Instead a reduction function R is produced to take a hash and produce a new message from it, where $h \in H$ and $p \in P$. R produces the mapping $h \to p$, this allows for an alternating chain of *message to hash to message...* to be produced.

To generate a hash-chain table[5], a random set of passwords from P is selected. Each of these is hashed, then the resulting hash is given to the reduction function, this is then hashed, this process is repeated k times, where k is the chain length. Instead of storing the entire chain, simply the starting point and the last password are stored.

$$\text{password} \xrightarrow{H} 2867\text{C}6\text{A}4 \xrightarrow{R} \text{tadcaacc} \cdots \text{nomzyaw} \xrightarrow{H} 3\text{b}5440\text{e}1 \xrightarrow{R} \text{bffbbecf}$$

Here only 'password' and 'bffbbecf' need to be stored to retrace every hash, and therefore generating password, seen while creating the table.

To use the table to attack a hash h, firstly the hash is passed through the reduction function, to produce a candidate password p. If this p appears in the set of endpoints of the table then the corresponding starting password is selected and the series of hash-reductions is applied until the original h is seen, the p which generated h is the plaintext preceding h in the chain. It isn't guaranteed that this search will be successful as multiple hashes are likely to reduce to the same p, this is illustrated in figure 5. If unsuccessful further hash-reducing is performed on h until a chain of length k is reached. If the hash is not discovered before reaching k, then the hash was never seen during the creation of the table and the search has failed.

As the number of chains created increases the likelihood of collisions increases also. A collision occurs when two chains produce the same value and continue hashing, this is likely to be due to password collision because of the reduction function, as

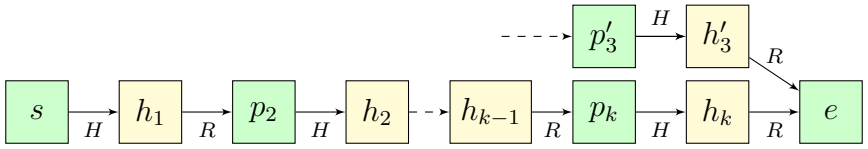

Figure 5: Visualisation of a hash-chain table, this demonstrates how finding the endpoint e does not necessarily reveal the password as multiple hashes are likely to reduce to the same password in the password space.

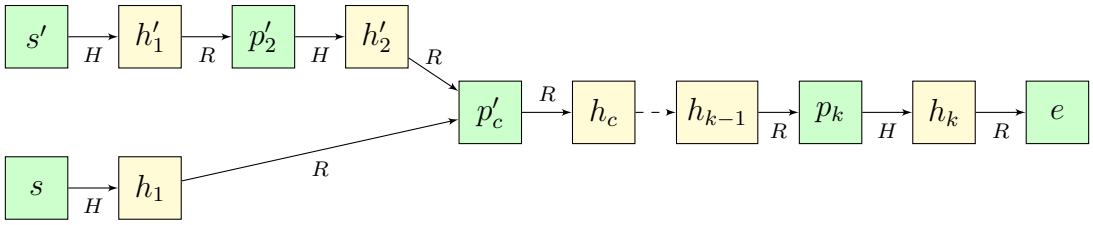

Figure 6: This is a demonstration of two chains colliding, from the point where R is applied to h'_2 no more useful work is performed by the chain starting at s'.

hashing functions are generally considered to be collision resistant.

Selecting an appropriate reduction function is a critical part of the design of a hash-chain table. It is important to know a lot about the possible password domain as constraining the search to this area will be more useful and generating a smaller table while covering the likely password space with less gaps, increasing the chance of finding the hash and its inverse. It is also important to generate the correct distribution of messages, it may be inefficient to create many short passwords if the space is up to 9 character for example. Needless to say designing an appropriate reduction function is difficult.

Rainbow Tables The largest shortcoming of simple hash-chain tables is that the number of collisions increases as the table size increases. To overcome this in 2003 Oechslin[9] proposed a method for using multiple reduction functions to drastically reduce the effect that a single collision has on the production of a chain in a table. This approach is called a rainbow table and is the dominant approach to launching pre-image attacks on hashing functions today.

A rainbow table has k different reduction functions $R_1 \rightarrow R_k$ and these are applied sequentially, alternating between hash and reduction application as in the hash-chain approach. This greatly reduces the damage caused by a collision. Collisions will still occur as shown in figure 7, but in the next step the application of a different reduction function will diverge the two chains again. It is possible to still

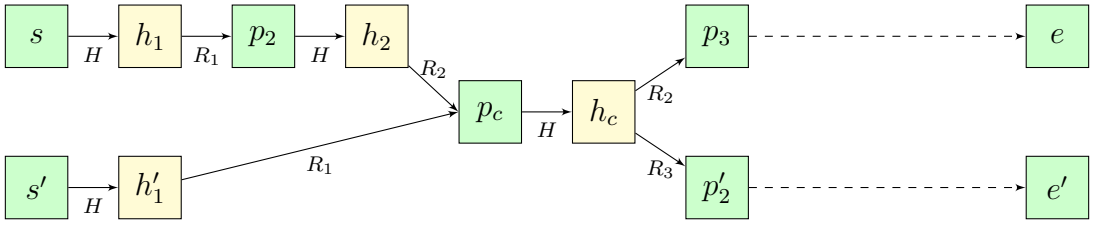

Figure 7: In a rainbow table as there are several reduction functions, as they are applied sequentially any branches merging will likely split on the application of the different reduction function.

have a collision which wastes the chain: if both chains merge on the same step, this will tie the two chains in to exploring the same state-space. This is much rarer than the collision case in the simple hash-chain table.

Using a rainbow table is still quite simple, given a hash, h to attack, the last reduction function, R_k is applied and the table is searched for a chain which has a matching endpoint. This is an attempt at guessing that the hash was the last hash the table saw. If none is found then the preceding reduction R_{k-1} is applied, followed by the hash and R_k. This is a guess that the h was the penultimate hash seen in each table chain. This continues until a hit is found in the set of endpoints. Once it is the starting point is selected and the chain is replayed until h is found, the preceding password is the password which hashed to h, as with the hash-chain table.

The original rainbow table paper demonstration, by Oechslin, constructed a rainbow table and accompanying program to crack Windows passwords. The program and wider principle behind rainbow tables has been subsequently used to demonstrate the tractability of pre-image attacks given that distributed resources are now available. Projects including `freerainbowtables.com` use many clients to distribute the work of generating hash chains. These projects have created a wealth of tables which these writers believe, though they do not break the hash's cryptographic one-way nature, do undermine the security of existing hashes and should raise questions about the continued use of algorithms such as MD5 and SHA-1.

4.3 Defending Against Tables

Pre-image attacks using tables are extremely effective at cracking hashes for passwords in the their password domain, however protecting stored passwords against these tables is simple.

Instead of storing just the hash, h, of the password p a salted hash is stored. Where s is some fixed-known string (salt) a salted hash can simple be made by concatenating the hash and salt,

$$h = H(p||s),$$

or

$$h = H(H(p)||s).$$

The salt need not be kept a secret as it should not be of little help when attempting to calculate a pre-image. It is feasible to create a hash table which takes into account a salt, so using a single salt is not a particularly safe approach as a single hash table could release the passwords to an entire database. Usually a salted hash and the salt are stored together, but importantly a salt will be different for each password, possibly the timestamp of a sign-up time for example. Ensuring the individual salts are different ensures that a fresh table would have to be created to crack each password and reduces any attack to a brute force approach, which is the best case.

An alternative defence would be to make the construction of such a table inconceivable. This can be achieved by a process called key-strectching. This process repeatedly applies the hashing function (often in excess of 1000) times to the password, this has the effect of increasing the amount of time it takes to compute each hash in the chain. This has a drastic impact on the total time taken to produce the table.This approach is susceptable to a very specific type of attack, one using custom hardware. Using FPGAs it is possible to construct hardware circuit which are able to unroll these repetition loops and therefore circumvent the added complexity added by the repetition stretching, this is a very specialised attack however and only viable when a lot of information is known about the encryption system. The custom hardware necessary would be very expensive and approaches such as **scrypt**[2] which as part of their process deliberately inflate memory usage have been futher developed to make specialist hardware and parallelism a less viable option for producing large amounts of hashes.

5 Side Channel Attacks

A side channel attack is an attack on the physical implementation of the cryptosystem. The main attacks are based on the fact there are efficient ways to do certain mathematical functions on a processor based on the values being processed, this is usually branching dependent on whether particular bits are 0 or 1. This approach has been widely applied on 3-DES schemes and only tried occasionally, with limited success to traditional hashes, such as MD5.

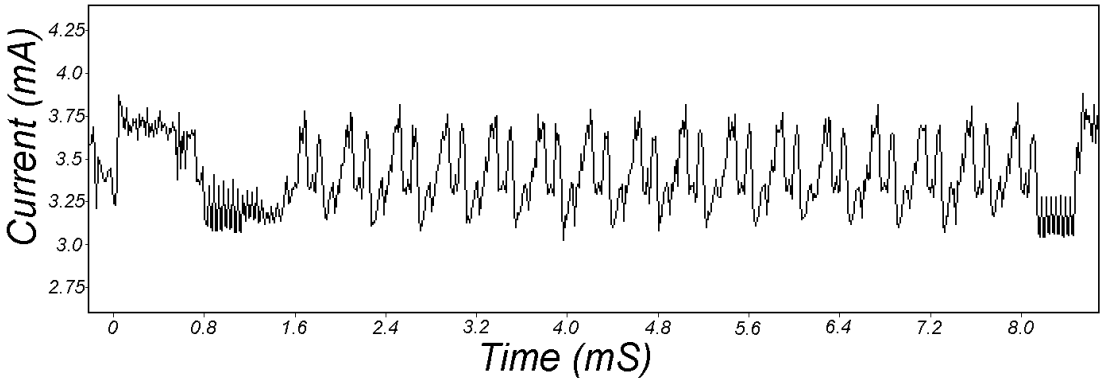

Figure 8: Trace showing a SPA of an entire DES operation.

5.1 Power Attacks

There are two types of Power Attacks. Using Simple Power Analysis(SPA) is simply watching the trace of the power flowing through the processor while it is performing the functions this shows the load on the processor and from that work out what action is being done. Peter Kocher outlined using DES in[13] showing that you can see the distinct rounds in the DES structure.

The other type is a Differential Power Analysis where many different checks are used and then a difference is applied to only show the 'significant' changes. Looking at DES in a round of the encryption the 8 S-Boxes each take 6 subkey bits and 6 bits of the R side and produce 4 bits of output as described in [6]. The selection function $D(C, b, K_s)$ is defined as the value of the bit $0 \leq b < 32$ of the DES L side before working out the 16th round for the cipher text C, where the 6 key bits entering the S box corresponding to the bit b are $0 \leq K_s < 2^6$. If we have the wrong key bits we will get the bit b right half of the time due to chance.

To use the DPA attack first takes m samples of cipher text and power traces $T_{1..m}[1..k]$ containing k samples each and evaluates D for them and seperates them into two groups g_0 and g_1 where the evaluations for b are 0 and 1 respectively. Then Δ_D is worked out for the means of the traces from g_0 and g_1. Then $\Delta_D[j]$ is the average over $C_{1..m}$ of the change due to the value of the selection function D on the power consumption at point j.

$$\Delta_D[j] = \frac{\sum_{i=1}^{m} D(C_i, b, K_s)T_i[j]}{\sum_{i=1}^{m} D(C_i, b, K_s)} - \frac{\sum_{i=1}^{m}(1 - D(C_i, b, K_s))T_i[j]}{\sum_{i=1}^{m}(1 - D(C_i, b, K_s))} \quad (1)$$

$$\approx 2(\frac{\sum_{i=1}^{m} D(C_i, b, K_s)T_i[j]}{\sum_{i=1}^{m} D(C_i, b, K_s)} - \frac{\sum_{i=1}^{m} T_i[j]}{m}) \quad (2)$$

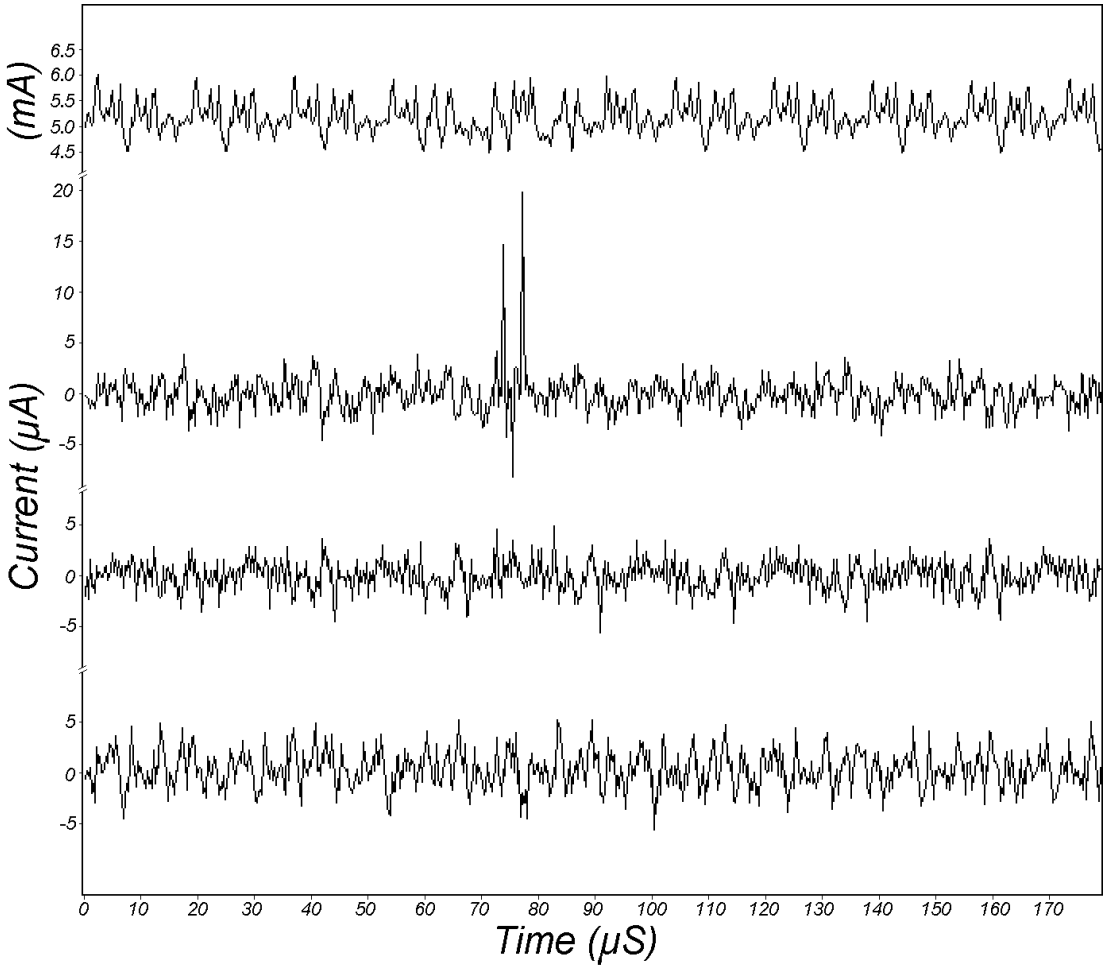

Figure 9: Traces showing the average trace then one with a correct guess of K_s and the bottom two with incorrect guesses with $m = 1000$

If K_s is correct however, the calculation of D will be correct for all the cipher text. As a result the $\Delta_D[j]$ will approach the effect of the target bit on the power consumption as $m \to \infty$.

The correct value of K_s can be identified from the spikes in the differential traces. Four values of b correspond to each S Box, providing validation of key block guesses. If you find all the eight K_s gives you the entire 48-bit round subkey. The remaining 8 key bits can be found either by exhaustive search or the attack could be focused on another round of the DES sequence. Triple DES keys can be found by analysing the outer DES operation, finding the key to decrypt the ciphertext and then attacking the next DES key. Shown in figure 9 are 4 traces of a reference trace followed by correct K_s and two incorrect one.

There is the belief that using there is a possibility to use DPA on a HMAC-MD5 based on a smartcard, due to the ease of control timing cycles and power input. This is outlined in [7]. However this turns out to be false and is blamed on the equipment used as this was set up with standard equipment not specialist equipment, it is suggested specialist hardware could improve the results. This would suggest, at the current time, that this type of attack best suited for to attacks on DES or RSA as described above.

5.2 Timing Attacks

There are quite a few different types of timing attacks which essentially take advantage of algorithms taking shortcuts to save computing power based on what it is processing. One of which is for computing $R = y^x \mod n$, where x is w bits long, which is used in RSA, is shown in 2 where X_k means the k^{th} bit of x as you can see if the bit of the exponent is 1 we have to compute an extra multiplication step which gives us the possibility of watching out for how long a pass through the algorithm takes which would give us if X_k is 0 or 1, which means we can rebuild the exponent. This is critical in RSA as y is the cipher text which can be eavesdropped upon and n is public. This is outlined in [3] as a possibility to crack RSA over the internet based on large number of observations to counteract the noise of the timing data.

Another type of timing attack is due to how checks of equality of iterables are usually lazily evaluated, as in they cut out when they find they are not equal as there is no gain from finishing the comparison due to no additional information gained. This has lead to the ability to see *how many bits we got right* this shows how problems can occur essentially outside of how secure any crypto algorithm is if there is side channel leak of information due to implementation either in the hardware running it or the exact way programmatically the algorithm is implemented.

Algorithm 2 Exponentiation By Squaring based off of the explanation in [13]

$S_0 \leftarrow 1$
for $k = 0 \rightarrow w - 1$ **do**
  if $X_k = 1$ **then**
    $R_k \leftarrow (s_k * y) \mod n$
  else
    $R_k \leftarrow s_k$
  end if
  $s_{k+1} \leftarrow R_k^2 \mod n$
end for
return (R_{w-1})

6 Conclusion

Hashing functions remain an integral part of many secure systems and are essential to signature verification and fast file comparison. The presence of hash collisions does not make a hash obselete it just means that users of the function should reconsider if the function is still fit for purpose.

> *Git uses SHA-1 in a way which has nothing at all to do with security. . . . it's just the best hash you can get. . . it's about the ability to trust your data* - Linus Tovalds on Git

While the security community continues to develop in tandem the latest and greatest hashing functions and publically displays weaknesses in existing algorithms, if developers are mindful of their algorithm choices hashes will continue being very useful tools and are unlikely to be made generally obselete by faster hardware or dedicated cryptanalysis.

References

[1] fips 180-1 - secure hash standard.

[2] Stronger key derivation via sequential memory-hard functions colin percival. *Access*, pages 1–16.

[3] Scott A Crosby, D A N S Wallach, and Rudolf H Riedi. Opportunities and Limits of Remote Timing Attacks. *System*, 12(3), 2009.

[4] Hans Dobbertin. Crytanalysis of MD5 Compress, 1996.

[5] Martin E Hellman. A Cryptanalytic Time - M e m o ry Trade-Off. I(4):401–406, 1980.

[6] P Kocher and J Jaffe. Differential power analysis. 1999.

[7] Jiří Kůr. Algorithms for differential power analysis of the cryptographic smart cards, 2006.

[8] Ralph Charles Merkle. Security, Authentication, And Public Key Systems. *Thesis*, (May), 1979.

[9] Philippe Oechslin. Making a Faster Cryptanalytic Time-Memory.

[10] R. Rivest. The MD5 Message-Digest Algorithm.

[11] Yu Sasaki, Go Yamamoto, and Kazumaro Aoki. Practical Password Recovery on an MD5 Challenge and Response. *Search*, pages 1–11, 2006.

[12] Marc Stevens. MD5 Considered Harmful Today • International team of researchers.

[13] Other Systems and Paul C Kocher. Timing Attacks on Implementations of.

[14] Xiaoyun Wang, Dengguo Feng, Xuejia Lai, and Hongbo Yu. Collisions for Hash Functions MD4, MD5, HAVAL-128 and RIPEMD. *Science*, 5:5–8, 2004.

[15] Tao Xie and Dengguo Feng. Construct MD5 Collisions Using Just A Single Block Of Message. page 61070228, 2010.

```
#!/usr/bin/python
import argparse, numpy as np
from bitstring import BitArray
from copy import copy
from math import floor, sin
parser = argparse.ArgumentParser(description='Calculate the MD5 hash of
parser.add_argument('hexstring')
args = parser.parse_args()

# will take an int and create a binary list, big endian
def int2bin(i, bitformat = None):
    bs = bin(i).lstrip('-0b')
    if bitformat is not None:
        bs = bs.zfill(bitformat)
    return map(int, bs)

def chunk(iterable, count):
    pos = 0
    end = len(iterable)
    while pos < end:
        yield iterable[pos:pos+count]
        pos+=count


def hex2bin(string, bitformat = None):
    try:
        return int2bin(int(string, 16), bitformat)
    except ValueError:
        return []

def hexle2hexbe(hexstring):
    ''' return the true hexstring represented by a loworder hex string''
    return ''.join(hexstring.split(' ')[::-1])


#original_bitstring = hex2bin(args.hexstring)
original_bitstring = BitArray(bin(int(args.hexstring, 16)))
modifiable_bitstring = BitArray(bin(int(args.hexstring, 16)))

# Step 1. Append Padding Bits
```

```
# append single '1'
modifiable_bitstring.append(1)
# append number of 0s needed to make the length of the message congruent
modifiable_bitstring.append([0]*((448 - (len(modifiable_bitstring) % 512

# should by 64 bits shy of being a multiple of 512
assert len(modifiable_bitstring)%512 == 448

# Step 2. Append Length
# length of original msg as little-endian 64-bit binary string
msglength64bit = BitArray(intle=len(original_bitstring), length=64)
modifiable_bitstring.append(msglength64bit)

# should be a multiple of 512 bits now
assert len(modifiable_bitstring)%512 == 0

# Step 3. Initialize MD Buffer
# each word is a 32-bit register
# intialised using hexadecimal, low-order bytes first
# Therefore: string...
A = BitArray(hex=hexle2hexbe('01 23 45 67'))
B = BitArray(hex=hexle2hexbe('89 ab cd ef'))
C = BitArray(hex=hexle2hexbe('fe dc ba 98'))
D = BitArray(hex=hexle2hexbe('76 54 32 10'))

# Step 4. Auxilliary functions brackets are for clarity
F = lambda X, Y, Z: (X & Y) | (~X & Z)
G = lambda X, Y, Z: (X & Z) | (Y & ~Z)
H = lambda X, Y, Z: X ^ Y ^ Z
I = lambda X, Y, Z: Y ^ (X | ~Z)

X = modifiable_bitstring
#    4294967296 == 2**32
T = [int(floor(4294967296 * abs(sin(i)))) for i in range(0, 65)]

AA = copy(A)
BB = copy(B)
CC = copy(C)
DD = copy(D)
```

```
def op(a, b, c, d, k, s, i, f):
    # a +F(b,c,d) + X[k] + T[i]
    inner_sum = (a.int + F(b,c,d).int + X[k] + T[i]) % 2^32
    inner_array = BitArray(uint=inner_sum, length=32)
    # <<< s
    inner_array.rol(s)
    # + b
    result = (b.int + inner_array.int)% 2^32
    # assign the result into A
    return BitArray(uint=result, length=32)

# round 1
A = op(A,B,C,D, k=0, s=7 , i=1, f=F)
D = op(D,A,B,C, k=1, s=12, i=2, f=F)
C = op(C,D,A,B, k=2, s=17, i=3, f=F)
B = op(B,C,D,A, k=3, s=22, i=4, f=F)

A = op(A,B,C,D, k=4, s=7 , i=5, f=F)
D = op(D,A,B,C, k=5, s=12, i=6, f=F)
C = op(C,D,A,B, k=6, s=17, i=7, f=F)
B = op(B,C,D,A, k=7, s=22, i=8, f=F)

A = op(A,B,C,D, k=8 , s=7 , i=9, f=F)
D = op(D,A,B,C, k=9 , s=12, i=10, f=F)
C = op(C,D,A,B, k=10, s=17, i=11, f=F)
B = op(B,C,D,A, k=11, s=22, i=12, f=F)

A = op(A,B,C,D, k=12, s=7 , i=13, f=F)
D = op(D,A,B,C, k=13, s=12, i=14, f=F)
C = op(C,D,A,B, k=14, s=17, i=15, f=F)
B = op(B,C,D,A, k=15, s=22, i=16, f=F)

# round 2
A = op(A,B,C,D, k=1 , s=5 , i=17, f=G)
D = op(D,A,B,C, k=6 , s=9 , i=18, f=G)
C = op(C,D,A,B, k=11, s=14, i=19, f=G)
B = op(B,C,D,A, k=0 , s=20, i=20, f=G)

A = op(A,B,C,D, k=5 , s=5 , i=21, f=G)
D = op(D,A,B,C, k=10, s=9 , i=22, f=G)
```

```
C = op(C,D,A,B, k=15, s=14, i=23, f=G)
B = op(B,C,D,A, k=4 , s=20, i=24, f=G)

A = op(A,B,C,D, k=9 , s=5 , i=25, f=G)
D = op(D,A,B,C, k=14, s=9 , i=26, f=G)
C = op(C,D,A,B, k=3 , s=14, i=27, f=G)
B = op(B,C,D,A, k=8 , s=20, i=28, f=G)

A = op(A,B,C,D, k=13, s=5 , i=29, f=G)
D = op(D,A,B,C, k=2 , s=9 , i=30, f=G)
C = op(C,D,A,B, k=7 , s=14, i=31, f=G)
B = op(B,C,D,A, k=12, s=20, i=32, f=G)

# round 3
A = op(A,B,C,D, k=5 , s=4 , i=33, f=H)
D = op(D,A,B,C, k=8 , s=11, i=34, f=H)
C = op(C,D,A,B, k=11, s=16, i=35, f=H)
B = op(B,C,D,A, k=14, s=23, i=36, f=H)

A = op(A,B,C,D, k=1 , s=4 , i=37, f=H)
D = op(D,A,B,C, k=4 , s=11, i=38, f=H)
C = op(C,D,A,B, k=7 , s=16, i=39, f=H)
B = op(B,C,D,A, k=10, s=23, i=40, f=H)

A = op(A,B,C,D, k=13, s=4 , i=41, f=H)
D = op(D,A,B,C, k=0 , s=11, i=42, f=H)
C = op(C,D,A,B, k=3 , s=16, i=43, f=H)
B = op(B,C,D,A, k=6 , s=23, i=44, f=H)

A = op(A,B,C,D, k=9 , s=4 , i=45, f=H)
D = op(D,A,B,C, k=12, s=11, i=46, f=H)
C = op(C,D,A,B, k=15, s=16, i=47, f=H)
B = op(B,C,D,A, k=2 , s=23, i=48, f=H)

# round 4
A = op(A,B,C,D, k=0 , s=5 , i=49, f=I)
D = op(D,A,B,C, k=7 , s=9 , i=50, f=I)
C = op(C,D,A,B, k=14, s=14, i=51, f=I)
B = op(B,C,D,A, k=5 , s=20, i=52, f=I)
```

```
A = op(A,B,C,D, k=12, s=5 , i=53, f=I)
D = op(D,A,B,C, k=3 , s=9 , i=54, f=I)
C = op(C,D,A,B, k=10, s=14, i=55, f=I)
B = op(B,C,D,A, k=1 , s=20, i=56, f=I)

A = op(A,B,C,D, k=8 , s=5 , i=57, f=I)
D = op(D,A,B,C, k=15, s=9 , i=58, f=I)
C = op(C,D,A,B, k=6 , s=14, i=59, f=I)
B = op(B,C,D,A, k=13, s=20, i=60, f=I)

A = op(A,B,C,D, k=4 , s=5 , i=61, f=I)
D = op(D,A,B,C, k=11, s=9 , i=62, f=I)
C = op(C,D,A,B, k=2 , s=14, i=63, f=I)
B = op(B,C,D,A, k=9 , s=20, i=64, f=I)

A = A + AA
B = B + BB
C = C + CC
D = D + DD
print "{} {} {} {}".format(A,B,C,D)
```