number for a DES key that will map the plaintext into that ciphertext.) Use your system to tell us what DES key mapped 0 to 5761b9ab9858b34b16. Free T-shirts to the first ten correct answers. 11. Show that DES encryption and decryption are identical except for the order of the 48-bit keys. Hint: running a round backwards is the same as running it forwards but with the halves swapped (see §3.3.4 A DES Round), and DES has a swap after round 16 when run forwards (see §3.3.1 DES Overview). 12. Verify the MixColumn result in Figure 3-25 by using the same method (in conjunction with Figure 3-28's table) to compute InvMixColumn of the MixColumn result and checking that you produce the MixColumn input. 13. Describe in detail the optimization of the Rijndael inverse round mentioned at the end of §3.5.6 Optimization. 14. Give an upper bound on the number of 258-octet states possible in RC4. 89 4 MODES OF OPERATION 4.1 INTRODUCTION We've covered how to encrypt a 64-bit block with DES or IDEA, or a 128-bit block with AES, but that doesn't really tell how to use these algorithms. For instance, sometimes messages to be encrypted don't happen to be exactly one block long. We also mentioned in §2.4.5 Integrity Check that it was possible, using a secret key encryption scheme, to generate a MAC (message authentication code). This chapter will attempt to tie up these various loose ends. 4.2 ENCRYPTING A LARGE MESSAGE How do you encrypt a message larger than 64 bits? There are several schemes defined in [DES81]. These schemes would be equally applicable to IDEA or any secret key scheme that encrypted fixedlength blocks, and no doubt one could come up with variant schemes as well. The ones defined in [DES81], and which we'll describe in detail, are: 1. Electronic Code Book (ECB) 2. Cipher Block Chaining (CBC) 3. k-Bit Cipher Feedback Mode (CFB) 4. k-Bit Output Feedback Mode (OFB) A newer scheme that might be important in the future is: 5. Counter Mode (CTR) 90 MODES OF OPERATION 4.2.1 4.2.1 Electronic Code Book (ECB) This mode consists of doing the obvious thing, and it is usually the worst method. You break the message into 64-bit blocks (padding the last one out to a full 64 bits), and encrypt each block with the secret key (see Figure 4-1). The other side receives the encrypted blocks and decrypts each block in turn to get back the original message (see Figure 4-2). There are a number of problems with this approach that don't show up in the single-block case. First, if a message contains two identical 64-bit blocks, the corresponding two blocks of ciphertext will be identical. This will give an eavesdropper some information. Whether it is useful or not depends on the context. We'll give an example where ECB would have a problem. Suppose that the eavesdropper knows that the plaintext is an alphabetically sorted list of employees and salaries being sent from management to payroll, tabularly arranged (see Figure 4-3). Further suppose that, as luck would have it, each line is exactly 64 bytes long, and the blocks happen to be divided in the salary field between the 1,000's and the 10,000's digit. Since identical plaintext blocks produce identical ciphertext blocks, not only can an eavesdropper figure out which sets of employees have identical salaries, but also which sets of employees have salaries in the same message break into blocks m1 m2 m3 m4 m5 m6 m7 m8 m9 encrypt with secret key EEEEEEEEE c1 c2 c3 c4 c5 c6 c7 c8 c9 Figure 4-1. Electronic Code Book Encryption c1 c2 c3 c4 c5 c6 c7 c8 c9 decrypt with secret key DDDDDDDDD m1 m2 m3 m4 m5 m6 m7 m8 m9 reassemble blocks message Figure 4-2. Electronic Code Book Decryption 4.2.2 ENCRYPTING A LARGE MESSAGE 91 $10,000 ranges. If he can guess a few relative salaries, he will have a pretty good idea of what all the salaries are from this "encrypted" message. Furthermore, if the eavesdropper is one of the employees, he can alter the message to change his own salary to match that of any other employee by copying the ciphertext blocks from that employee to the corresponding blocks of his own entry. Even a human looking at the resulting message would see nothing awry. So, ECB has two serious flaws. Someone seeing the ciphertext can gain information from repeated blocks, and someone can rearrange blocks or modify blocks to his own advantage. As a result of these flaws, ECB is rarely used to encrypt

messages. 4.2.2 Cipher Block Chaining (CBC) CBC is a method of avoiding some of the problems in ECB. Using CBC, even if the same block repeats in the plaintext, it will not cause repeats in the ciphertext. First we'll give an example of how this might be accomplished. Our example is not CBC, but it helps for understanding CBC. Generate a 64-bit random number ri for each plaintext block mi to be encrypted. ⊕ the plaintext block with the random number, encrypt the result, and transmit both the unencrypted random Name Position Salary Adams, John President 78,964.31 Bush, Neil Accounting Clerk 623,321.16 Hoover, J. Edgar Wardrobe Consultant 34,445.22 Stern, Howard Affirmative Action Officer 38,206.51 Woods, Rosemary Audiovisual Supervisor 21,489.15 ||||||||| Block boundaries Figure 4-3. Payroll Data m1 m2 m3 m4 m5 m6 generate random numbers r1 ⊕ r2 ⊕ r3 ⊕ r4 ⊕ r5 ⊕ r6 ⊕ encrypt with secret key EEEEEE c1 c2 c3 c4 c5 c6 Figure 4-4. Randomized Electronic Code Book Encryption 92 MODES OF OPERATION 4.2.2 number ri and the ciphertext block ci (see Figure 4-4). To decrypt this, you'd decrypt all the cis, and for each ci, after decrypting it, you'd ⊕ it with the random number ri. The main problem with this scheme is efficiency. It causes twice as much information to be transmitted, since a random number has to be transmitted along with each block of ciphertext. Another problem with it is that an attacker can rearrange the blocks and have a predictable effect on the resulting plaintext. For instance, if r2|c2 were removed entirely it would result in m2 being absent in the decrypted plaintext. Or if r2|c2 were swapped with r7|c7, then m2 and m7 would be swapped in the result. Worse yet, an attacker knowing the value of any block mn can change it in a predictable way by making the corresponding change in rn. Now we can explain CBC. CBC generates its own "random numbers". It uses ci as ri+1. In other words, it takes the previous block of ciphertext and uses that as the random number that will be ⊕'d into the next plaintext. To avoid having two plaintext messages that start the same wind up with the same ciphertext in the beginning, CBC does select one random number, which gets ⊕'d into the first block of plaintext, and transmits it along with the data. This initial random number is known as an IV (initialization vector). Decryption is simple because ⊕ is its own inverse. Since the cost of the ⊕ is trivial compared to the cost of an encryption, CBC encryption has the same performance as ECB encryption except for the cost of generating and transmitting the IV. m1 m2 m3 m4 m5 m6 IV ⊕⊕⊕⊕⊕⊕ encrypt with secret key EEEEEE c1 c2 c3 c4 c5 c6 Figure 4-5. Cipher Block Chaining Encryption m1 m2 m3 m4 m5 m6 IV ⊕⊕⊕⊕⊕⊕ decrypt with secret key DDDDDD c1 c2 c3 c4 c5 c6 Figure 4-6. Cipher Block Chaining Decryption 4.2.2.1 ENCRYPTING A LARGE MESSAGE 93 In many cases the security of CBC would not be adversely affected by omitting the IV (or, equivalently, using the value 0 as the IV). However, we'll give one example where it would matter. Suppose the encrypted file of employees and salaries is transmitted weekly. If there were no IV, then an eavesdropper could tell where the ciphertext first differed from the previous week, and therefore perhaps determine the first person whose salary had changed. Another example is where a general sends information each day saying continue holding your position. The ciphertext will be the same every day, until the general decides to send something else, like start bombing. Then the ciphertext would suddenly change, alerting the enemy. A randomly chosen IV guarantees that even if the same message is sent repeatedly, the ciphertext will be completely different each time. Finally, a randomly chosen IV prevents attackers from supplying chosen plaintext to the underlying encryption algorithm even if they can supply chosen plaintext to the CBC. 4.2.2.1 CBC Threat 1—Modifying Ciphertext Blocks Using CBC does not eliminate the problem of someone modifying the message in transit; it does change the nature of the threat. Attackers can no longer see repeated values and simply copy or move ciphertext blocks in order to, say, swap the janitor's salary with the salary of the VP of marketing. But they certainly can still modify the ciphertext. What would happen if they changed a block of the ciphertext, say the value of ciphertext block

cn? cn gets ⊕'d with the decrypted cn+1 to yield mn+1, so changing cn has a predictable effect on mn+1. For instance, changing bit 3 of cn changes bit 3 of mn+1. cn also gets decrypted, and then ⊕'d with cn−1 to produce mn. Our attacker cannot know what a particular new value of cn would decrypt to, so changing cn will most likely garble mn to some random 64-bit value. For example, let's say our attacker knows that the plaintext corresponding to a certain byte range in the ciphertext is her personnel record: Let's say Jo wants to increase her salary by 20K. In this case she knows that the final byte of m7 is the ten-thousands digit of her salary. The bottom three bits of the ASCII for 5 is 101. To give herself a raise of 20K, she merely has to flip the penultimate bit of c6. Since c6 gets ⊕'d into the decrypted c7 (c7 has not been modified, so the decrypted c7 will be the same), the result will be the same as before, i.e. the old m7, but with the penultimate bit flipped, which will change the ASCII 5 into a 7. Unfortunately for Jo, as a side-effect a value she will not be able to predict will appear in her department field, since she cannot predict what the modified c6 will decrypt to, which will affect m6: Tacker, Jo A System Security Officer 54,122.10 ||||||||| 94 MODES OF OPERATION 4.2.2.2 A human who reads this report and issues checks is likely to suspect something is wrong; however, if it's just a program, it would be perfectly happy with it. And a bank would most likely take the check even if some unorthodox information appeared in what to them is a comment field. In the above example, Jo made a change she could control in one block at the expense of getting a value she could neither control nor predict in the preceding block. 4.2.2.2 CBC Threat 2—Rearranging Ciphertext Blocks Suppose Jo knows the plaintext and corresponding ciphertext of some message. So she knows m1, m2,...mn. And she knows IV, c1, c2,...cn. She will also know what each of the ci decrypt to, since the decrypted version of ci is ci−1⊕ mi (see Figure 4-7). Given this knowledge, she can consider each of the ci as a building block and construct a ciphertext stream using any combination of ci and she will be able to calculate what the corresponding plaintext would be. How could this be useful? Well, admittedly it's stretching a bit, though people in the security field enjoy stretching. One of the ways to combat threat 1 is to attach a CRC to the plaintext before encrypting it with CBC. That way if Jo modifies any of the ciphertext blocks, the CRC will not match the message and a computer can verify that the message has been tampered with. Still following? Well, suppose a 32-bit CRC is chosen. Then there is a 1 in 232 chance that the CRC will happen to be correct after the message is tampered with. Suppose Jo doesn't care what she garbles the message to. She just wants the garbage accepted by the computer on the other side, knowing that it will check the CRC. Jo can try constructing lots of ciphertext streams out of c1, c2,...cn, calculate the resulting plaintext from each one, and then test the resulting plaintext to see if the CRC comes out correct. On the average she will only have to try 231 arrangements before being able to find a message that will have a correct CRC. Tacker, Jo A System Security Of#f8Ts9(* 74,122.10 ||||||||| m1 m2 m3 m4 m5 m6 IV ⊕⊕⊕ ⊕ ⊕ ⊕ decrypt with secret key DDD D D D c1 c2 c3 c4 c5 c6 Figure 4-7. Cipher Block Chaining Decryption 4.2.3 ENCRYPTING A LARGE MESSAGE 95 What harm could Jo do by garbling a message into something she could not control the contents of, but would be accepted by the computer on the other side? Perhaps Jo is just being malicious, and wants to destroy some data being loaded across the network. But there is a subtle way in which Jo can actually control the garbling of the message to a small extent. Suppose she moves contiguous blocks; for instance, she might move cn and cn+1 to some other place. Then the original mn+1 will appear in that other position. Perhaps if mn+1 contains the president's salary, Jo can swap blocks so that her salary will be changed to his, but then she'll be forced to pretty much totally garble the entire rest of the file in order to find an arrangement of blocks that will result in a correct CRC. To prevent Jo from being able to rearrange blocks to find something that will have a correct CRC, a 64-bit CRC could be used. This would certainly suffice if the only attack on the

CRC inside CBC encoding was a brute force attack such as we described. There have been some interesting attacks suggested in the theoretical community [JUEN84] though these attacks are not practical. An encryption mode that protects both the confidentiality and the authenticity of a message using a single cryptographic pass over the data has been the holy grail of cryptographic protocol design for many years, with many proposals subsequently discredited. There are some new ones that look promising (for example, see §4.3.5 Offset Codebook Mode (OCB)), but none has been formally standardized. 4.2.3 Output Feedback Mode (OFB) Output feedback mode is a stream cipher. Encryption is performed by ⊕ing the message with the one-time pad generated by OFB. Let's assume that the stream is created 64 bits at a time. To start, a random 64-bit number is generated, known as the IV (as in CBC mode). Let's call that b0. Then b0 is encrypted (using the secret key) to get b1, which is in turn encrypted to get b2, and so forth. The resulting one-time pad is b0|b1|b2|b3|.... To encrypt a message, merely ⊕ it with as many bits of b0|b1|b2|b3|... as necessary. The result is transmitted along with the IV. The recipient computes the same one-time pad based on knowledge of the secret key and the IV. To decrypt the message, the recipient merely ⊕s it with as many bits of b0|b1|b2|b3|... as necessary. The advantages of a system like this are: 1. The one-time pad can be generated in advance, before the message to be encrypted is known. When the message arrives to be encrypted, no costly cryptographic operations are needed. Instead only ⊕ is required, and ⊕ is extremely fast. 96 MODES OF OPERATION 4.2.4 2. If some of the bits of the ciphertext get garbled, only those bits of plaintext get garbled, as opposed to in CBC mode where if cn is garbled then mn will be completely garbled and the same portion of mn+1 as was garbled in cn will be garbled. 3. A message can arrive in arbitrarily sized chunks, and each time a chunk appears, the associated ciphertext can be immediately transmitted. In contrast, with CBC, if the message is arriving a byte at a time, it cannot be encrypted until an entire 64-bit block of plaintext is available for encryption. This results in either waiting until 7 more bytes arrive, or padding the plaintext to a multiple of 8 bytes before encrypting it, which yields more ciphertext to be transmitted and decrypted. The disadvantages of OFB are: 1. If the plaintext and ciphertext are known by a bad guy, he can modify the plaintext into anything he wants by simply ⊕ing the ciphertext with the known plaintext, and ⊕ing the result with whatever message he wants to transmit. The FIPS document specifies OFB as being capable of being generated in k-bit chunks. The description above would be equivalent to 64-bit OFB as documented in [DES81]. The way k-bit OFB (see Figure 4-10) works is as follows. The input to the DES encrypt function is initialized to some IV. If the IV is less than 64 bits long, it is padded with 0's on the left (most significant portion). The output will be a 64-bit quantity. Only k bits are used, which happen to be specified in [DES81] as the most significant k bits. Cryptographically, any k bits would do, though it is nice to standardize which bits will be selected so that implementations will interoperate. So, we've generated the first k bits of the one-time pad. Now the same k bits are shifted into the rightmost portion of the input, and what was in that register is shifted k bits to the left. Now k more bits of one-time pad are selected, as before. 4.2.4 Cipher Feedback Mode (CFB) CFB (see Figure 4-9) is very similar to OFB, in that k bits at a time are generated and ⊕'d with k bits of plaintext. In OFB the k bits that are shifted into the register used as the input to the DES encrypt are the output bits of the DES encrypt from the previous block. In contrast, in CFB, the k bits shifted in are the k bits of ciphertext from the previous block. So in CFB the one-time pad cannot be generated before the message is known (as it can be in OFB). It is sensible to have k-bit CFB with k something other than 64, and in particular k = 8 makes sense. With OFB or CBC, if characters are lost in transmission, or extra characters are added to the ciphertext stream, the entire rest of the transmission is garbled. With 8-bit CFB, as long as the error is an integral number of bytes, things will resynchronize. If a byte is lost in

transmission, one byte of plaintext will be lost and the next 8 bytes will be garbled. However, after that, the plaintext will 4.2.4 ENCRYPTING A LARGE MESSAGE 97 decrypt properly. If a byte is added to the ciphertext stream, then a byte of garbage will be added to the plaintext stream, and the following 8 bytes of plaintext will be garbled, but after that it will decrypt properly. It does have the disadvantage that every byte of input requires a DES operation. CFB-encrypted messages are somewhat less subject to tampering than either CBC or OFB. With 8-bit CFB it is possible for an attacker to change any individual byte of plaintext in a predictable way at the cost of unpredictably garbling the next 8 bytes. With 64-bit CFB it is possible for an attacker to modify any 64-bit block in a predictable way at the cost of unpredictably garbling the next 64-bit block. There is no block-rearranging attack as with CBC, though whole sections of the message can be rearranged at the cost of garbling the splice points. IV K E K E K E discarded discarded discarded k bits k bits k bits m1 ⊕ m2 ⊕ m3 ⊕ k bits k bits k bits c1 c2 c3 Figure 4-8. k-bit OFB IV K E K E K E discarded discarded discarded k bits k bits k bits m1 ⊕ m2 ⊕ m3 ⊕ k bits k bits k bits c1 c2 c3 Figure 4-9. k-bit CFB 98 MODES OF OPERATION 4.2.5 In principle, CFB need not be used on a byte boundary. It could iterate for any integer number of bits up to a full block. In practice, it's only used on byte boundaries or as a full block. When done in full-block mode, it has performance comparable to ECB, CBC, and OFB; it has the OFB advantage of being able to encrypt and send each byte as it is known; it lacks OFB's ability to compute a substantial amount of pad ahead; and it detects alterations better than OFB but not as well as CBC. 4.2.5 Counter Mode (CTR) Counter mode is similar to Output Feedback Mode in that a one-time pad is generated and ⊕'d with the data. It is different in that instead of chaining by encrypting each one-time pad block to get the next one, OFB increments the IV and encrypts the result to get successive blocks of the one-time pad. The main advantage of counter mode is that, like OFB, the cryptography can be pre-computed and encryption is simply an ⊕, but like CBC, you can decrypt the message starting at any point rather than being forced to start at the beginning. This makes counter mode ideal for applications like encrypting a randomly accessed file. The subset of the data you will need and the order in which you will need it is unpredictable. Like OFB, CTR loses security if different data is encrypted with the same key and IV. An attacker could get the ⊕ of two plaintext blocks by taking the ⊕ of the two corresponding ciphertext blocks. IV IV+1 IV+2 K E K E K E m1 ⊕ m2 ⊕ m3 ⊕ c1 c2 c3 Figure 4-10. Counter Mode (CTR) 4.3 GENERATING MACS 99 4.3 GENERATING MACS A secret key system can be used to generate a cryptographic checksum known as a MAC (message authentication code). A synonym for MAC is MIC (Message Integrity Code). The term MAC seems to be more popular though MIC is sometimes used, for instance in PEM (see §18.15.3 MICONLY or MIC-CLEAR, Public Key Variant). While CBC, CFB, OFB, and CTR, when properly used, all offer good protection against an eavesdropper deciphering a message, none offers good protection against an eavesdropper who already knows the contents of the plaintext message modifying it undetected. A standard way for protecting against undetected modifications is to compute the CBC but send only the last block along with the plaintext message. This last block is called the CBC residue. In order to compute the CBC residue, you have to know the secret key. If an attacker modifies any portion of a message, the residue will no longer be the correct value (except with probability 1 in 264). And the attacker will not be able to compute the residue for the modified message without knowing the secret key. In this case, the picture for the recipient of the message is the same as for the sender. The recipient computes the CBC residue based on the plaintext message and sees whether it matches the one sent. If it does, someone who knew the key computed the MAC on that message. In many contexts, messages are not secret but their integrity must be ensured. In such contexts it is perfectly appropriate to transmit unencrypted data, plus a CBC residue. Interbank transfers are a traditional example (there may be a desire for secrecy as well,

but it is dwarfed by the requirement for accuracy). But more commonly, there is a desire to encrypt messages for both privacy and integrity. This can be done with a single encryption operation if the message is a single block. What is the equivalent transformation on a multiblock message? (Keep reading to find out!) m1 m2 m3 m4 m5 m6 ⊕⊕⊕⊕⊕ encrypt with secret key EEEEEE c1 c2 c3 c4 c5 CBC residue Figure 4-11. Cipher Block Chaining Residue 100 MODES OF OPERATION 4.3.1 4.3.1 Ensuring Privacy and Integrity Together To summarize, if we have a message and we want to ensure its privacy, we can CBC-encrypt the message. If we have a message and we want to ensure its integrity, then we can send the CBC residue along with the message. It is natural to assume that if we want to be secure against both modification and eavesdropping that we ought to be able to do something like Figure 4-12. Well, that can't be right. That consists of sending the CBC encrypted message and just repeating the final block. Anyone tampering with the CBC encrypted message could simply take the value of the final block they wanted to send, and send it twice. So sending the CBC residue in addition to the CBC encrypted message can't enhance security. Maybe simply sending the CBC encrypted message gives privacy and integrity protection. Well, when we say integrity protection we mean we'd like the receiving computer to automatically be able to tell if the message has been tampered with. With CBC alone there is no way to detect tampering automatically, since CBC is merely an encryption technique. Any random string of bits will decrypt into something, and the final 64 bits of that random string will be a "correct" CBC residue. So it is easy for anyone to modify the ciphertext, and a computer on the other side will decrypt the result, come up with garbage, but have no way of knowing that the result is garbage. If the message was an English message, and a human was going to look at it, then modification of the ciphertext would probably get detected, but if it is a computer merely loading the bits onto disk, there is no way for the computer to know, with this scheme, that the message has been modified. OK, how about computing the CBC residue, attaching that to the plaintext, and then doing a CBC encryption of the concatenated quantity (Figure 4-13)? It turns out that that doesn't work either. The last block is always the encryption of zero, since the ⊕ of anything with itself is always zero. An extra block that doesn't depend on the message can't offer any integrity protection. Let's try one more. Suppose we compute a non-cryptographic checksum (e.g., a CRC) of the blocks of the message, append that to the end, and encrypt the whole thing (Figure 4-14). This almost works. Subtle attacks are known if the CRC is short. Longer non-cryptographic checksums are suspect. m1 m2 m3 m4 m5 m6 IV ⊕⊕⊕⊕⊕⊕ encrypt with secret key EEEEEE c1 c2 c3 c4 c5 c6 CBC residue Figure 4-12. Cipher Block Chaining Encryption plus CBC Residue 4.3.2 GENERATING MACS 101 So, what can we do? It is generally accepted as secure to protect the privacy of a message with CBC encryption and integrity with a CBC residue as long as the two are computed with different keys. Unfortunately, this requires twice the cryptographic power of encryption alone. Various shortcuts have been proposed and even deployed, but generally they have subtle cryptographic flaws. Whether those flaws are serious depends on the application and the cleverness of the attacker. It's instructive to look at some of the techniques. 4.3.2 CBC with a Weak Cryptographic Checksum Jueneman proposed [JUEN84, JUEN85] that since using non-cryptographic checksums inside CBC did not appear to be secure, and since quality cryptographic checksums were expensive to compute, perhaps a "weak" cryptographic checksum inside a CBC would provide good security on the basis that the computational complexity of breaking the weak checksum would be multiplied by the limitations of having to do it under the constraints of the CBC. He proposed a weak cryptographic checksum for this use. There is no reason to believe that this approach would not be effective, but it has not caught on. Ironically, Kerberos V4 (see §11.10 Encryption for Integrity Only) uses a variant of his weak m1 m2 m3 m4 m5 m6 c6 IV ⊕⊕⊕⊕⊕⊕⊕ EEEEEEE c1 c2 c3 c4 c5 c6 c7 Figure 4-13.

Cipher Block Chaining Encryption of Message with CBC Residue m1 m2 m3 m4 m5 m6 crc IV ⊕⊕⊕⊕⊕⊕⊕ EEEEEEE c1 c2 c3 c4 c5 c6 c7 Figure 4-14. Cipher Block Chaining Encryption of Message with CRC 102 MODES OF OPERATION 4.3.3 checksum for integrity protection outside of an encrypted message, and there is no evidence that anyone has broken it. 4.3.3 CBC Encryption and CBC Residue with Related Keys It is generally accepted as secure to compute a message integrity code by computing a CBC residue and then encrypting the message using an independently chosen key. A trick used by Kerberos V5 is to use a modified version of the key for one of the operations. (Switching one bit should be sufficient, but Kerberos instead ⊕s the key with the constant F0F0F0F0F0F0F0F016. This has the nice properties of preserving key parity and never transforming a non-weak key into a weak key. See §3.3.6 Weak and Semi-Weak Keys.) There are no known weaknesses in the approach of having one key be mathematically related to the other (as opposed to simply picking two random numbers for keys), but few advantages either. In general it's no harder to distribute a pair of keys than a single one, and having the keys be mathematically related saves no computational effort. The only advantage of deriving one key from the other is to get both privacy and integrity protection when a mechanism is in place for distributing only a single key. 4.3.4 CBC with a Cryptographic Hash Another approach is to put a cryptographic hash of the message—typically 128 bits—inside, and CBC encrypt the whole thing. This is probably secure (though it hasn't gotten much use or scrutiny, since modern schemes use a keyed hash). It requires two cryptographic passes (just like doing CBC encryption and CBC residue with different keys), but it is more efficient than doing CBC twice if the hash function is faster than the encryption algorithm. 4.3.5 Offset Codebook Mode (OCB) OCB is one of several modes that get both encryption and integrity protection while making only a single cryptographic pass over the data. This and similar schemes are too new to have stood the test of time, and there are patent issues hanging over them, but it seems likely that one or more will eventually become the standard way to achieving this effect. 4.4 MULTIPLE ENCRYPTION DES 103 4.4 MULTIPLE ENCRYPTION DES The generally accepted method of making DES more secure through multiple encryptions is known as EDE (for encrypt–decrypt–encrypt) or 3DES. Actually, any encryption scheme might be made more secure through multiple encryptions. Multiple encryption, though, has specifically been discussed in the industry in order to arrive at a "standard" means of effectively increasing DES's key length. EDE could as easily be done with, say, IDEA. It is more important with DES than with IDEA, though, because of DES's key length (or lack thereof). Remember that a cryptographic scheme has two functions, known as encrypt and decrypt. The two functions are inverses of each other, but in fact each one takes an arbitrary block of data and garbles it in a way that is reversed by the other function. So it makes sense to perform decrypt on the plaintext as a method of encrypting it, and then perform encrypt on the result as a method of getting back to the plaintext again. It might be confusing to say something like encrypt with decryption, so we'll just refer to the two functions as E and D. How to do multiple encryption is not completely obvious, especially since there is also the problem of turning block encryption into stream encryption. The standard method for using EDE is: 1. Two keys are used: K1 and K2. 2. Each block of plaintext is subjected to E with K1, then D with K2, and then E with K1. The result is simply a new secret key scheme—a 64-bit block is mapped to another 64-bit block. Decryption simply reverses the operation. 3. CBC is used to turn the block encryption scheme resulting from step 2 into a stream encryption (Figure 4-15). Now we'll discuss why 3DES is defined this way. There are various choices that could have been made: K1 K2 K1 m EDE c K1 K2 K1 c DED m 104 MODES OF OPERATION 4.4 1. Three encryptions were chosen. It could have been 2 or 714. Is three the right number? 2. Why are the functions EDE rather than, say, EEE or EDD? 3. Why is the cipher block chaining done on the outside rather than on the inside? Doing CBC on the inside (Figure 4-16) means completely

encrypting the message with CBC, then completely decrypting the result with CBC using a second key, and then completely CBCencrypting the message again. Doing CBC on the outside (Figure 4-15) means performing the DES encryptions and decryption on each block, but only doing the CBC once. m1 m2 m3 m4 m5 m6 IV ⊕⊕⊕⊕⊕⊕ encrypt with secret key K1 decrypt with secret key K2 encrypt with secret key K1 EEEEEE DDDDDD EEEEEE c1 c2 c3 c4 c5 c6 Figure 4-15. EDE with CBC on the Outside (3DES) m1 m2 m3 m4 m5 m6 IV ⊕⊕⊕⊕⊕⊕ encrypt with secret key K1 decrypt with secret key K2 EEEEEE DDDDDD ⊕⊕⊕⊕⊕ encrypt with secret key K1 EEEEEE c1 c2 c3 c4 c5 c6 Figure 4-16. EDE with CBC on the Inside 4.4.1 MULTIPLE ENCRYPTION DES 105 4.4.1 How Many Encryptions? Let's assume that the more times the block is encrypted, the more secure it is. So encrypting 749 times would therefore be some amount more secure than encrypting 3 times. The problem is, it is expensive to do an encryption. We don't want to do any more encryptions than are necessary for the scheme to be really secure. 4.4.1.1 Encrypting Twice with the Same Key Suppose we didn't want to bother reading in two keys. Would it make things more secure if we encrypted twice in a row with the same key? This turns out not to be much more secure than single encryption with K, since exhaustive search of the keyspace still requires searching only 256 keys. Each step of testing a key is twice as much work, since the attacker needs to do double encryption, but a factor of two for the attacker is not considered much added security, especially since the good guys have their work doubled with this scheme as well. (How about E followed by D using the same key? That's double the work for the good guy and no work for the bad guy, so that's generally not considered good cryptographic form.) 4.4.1.2 Encrypting Twice with Two Keys If encrypting twice, using two different keys, were as secure as a DES-like scheme with a key length of 112 bits, then encrypting twice would have sufficed. However, it isn't, as we will show. We'll use two DES keys, K1 and K2, and encrypt each block twice, first using key K1 and then using key K2. For the moment, let's ignore any block chaining schemes and assume we're just doing block encryption. Is this as cryptographically strong as using a double-length secret key (112 bits)? The reason you might think it would be as strong as a 112-bit key is that the straightforward brute-force attack would have to guess both K1 and K2 in order to determine if a particular plaintext block encrypted to a particular ciphertext block. However, there is a less straightforward attack that breaks double-encryption DES in roughly twice the time of a brute-force breaking of single-encryption DES. The attack is not particularly practical, but the fact that it exists makes double encryption sufficiently suspect that it's not generally done. The threat involves the following steps: plaintext ciphertext K K → → plaintext ciphertext → → K K 1 2 106 MODES OF OPERATION 4.4.1.3 1. Assume you have a few ⟨plaintext,ciphertext⟩ pairs ⟨m1,c1⟩, ⟨m2,c2⟩, ⟨m3,c3⟩ where ci was derived from doubly encrypting mi with K1 and K2. You want to find K1 and K2. 2. First make Table A with 256 entries, where each entry consists of a DES key K and the result r of applying that key to encrypt m1. Sort the table in numerical order by r. 3. Now make Table B with 256 entries, where each entry consists of a DES key K and the result r of applying that key to decrypt c1. Sort the table in numerical order by r. 4. Search through the sorted lists to find matching entries, ⟨KA,r⟩ from table A and ⟨KB,r⟩ from Table B. Each match provides KA as a candidate K1 and KB as a candidate K2 because KA encrypts m1 to r and KB encrypts r to c1. 5. If there are multiple intermediate values (which there almost certainly will be), test the candidate K1 and K2 against m2 and c2. If you've tested all the candidate ⟨K1,K2⟩ pairs and multiple of them work on m2 and c2, then try m3 and c3. The correct pair will always work, of course, and an incorrect pair will almost certainly fail to work on any particular ⟨mi ,ci⟩ pair. (See below for exact numbers, if you care.) This attack is not really practical, since a table of size 256 is a bit daunting. The existence of the attack, though, is enough reason to bother doing triple DES encryption. It may be that double encryption would be

good enough, but since triple encryption isn't that much harder, it will save you from defending yourself to management when they ask you why you're doing double encryption when they've heard that double encryption isn't secure. How many matches should you expect to find after searching the two tables? Well, there are 264 possible blocks, and only 256 table entries in each table (because there are only 256 keys). Therefore each 64-bit block has only a 1 in 256 chance of appearing in each of the tables. Of the 256 blocks that appear in Table A, only 1/256 of them also appear in Table B. That means that there should be about 248 entries which appear in both tables. One of those corresponds to the correct ⟨K1,K2⟩ pair and the others are imposters. We'll test them against ⟨m2,c2⟩. If ⟨K1,K2⟩ is an imposter, the probability that D(c2,K2) will equal E(m2,K1) is about 1/264. There are about 248 imposters, so the probability of one of them satisfying D(c2,K2) = E(m2,K1) is about 248/264, or roughly 1 in 216. Each test against an additional ⟨mi ,ci⟩ reduces the probability by a factor of 264, so the probability that there will still be false matches after trying three ⟨m, c⟩ pairs is about 1/280. 4.4.1.3 Triple Encryption with only Two Keys 3DES does triple encryption. Why does 3DES use only two keys? Presumably, security cannot possibly be enhanced because K1 is used twice rather than using a third key. People believe using K1 twice in this way is sufficiently secure that there's no reason to bother generating, transmitting, and storing a third key. For avoiding the brute-force exhaustive key space attack, 112 bits of key is con- 4.4.2 MULTIPLE ENCRYPTION DES 107 sidered sufficient, and no attacks other than the straightforward brute-force search is known for EDE. Some systems do implement 3DES with three independent keys, but this is not the standard. An esoteric reason for using only two keys is that in some applications (like §5.2.4.1 UNIX Password Hash) an important property of a cryptosystem is that given a ⟨plaintext, ciphertext⟩ pair, it is impractical to find any key that maps the plaintext to the ciphertext. (With 64-bit blocks and 112-bit keys, there will be lots of such keys.) Using EDE with three keys, it is straightforward to find a triple of keys that maps a given plaintext to a given ciphertext (Homework Problem 4). There is no known practical way of finding such a triple with K1 = K3. Why is K2 used in decrypt mode? Admittedly, it is no more trouble to run DES in either mode, and either way gives a mapping. DES would be just as good always done backwards (i.e., swap encrypt and decrypt). One reason for the choice of EDE is that an EDE system with K1 = K2 is equivalent to simple DES, so that such a system can interoperate with a simple DES system. Another reason to do EDE rather than EEE is that if E is followed by E, then the initial permutation before the second E will reverse the final permutation after the first E. As we said in §3.3.2 The Permutations of the Data, there is no security value in the initial or final permutation, at least when a single encryption is done. However, there might conceivably be some security gained by having a permutation done between encryption steps. Doing E followed by D results in doing the final permutation twice, once after E and once at the beginning of D. Doing the final permutation twice results in some permutation (Homework Problem 7). Likewise, doing D followed by E results in doing the initial permutation twice. 4.4.2 CBC Outside vs. Inside To review, the 3DES commonly used in the industry (and defined in §4.4 Multiple Encryption DES) does CBC on the outside, which means that each block is triply encrypted, and the CBC is done on the triply encrypted block (see Figure 4-15). The alternative would be to completely encrypt the message with K1 and CBC, and then take the result and completely decrypt it with K2 and CBC, and then take the result and completely encrypt it with K1 (see Figure 4-16). What are the implications of this choice? As shown in §4.2.2.1 CBC Threat 1—Modifying Ciphertext Blocks, with CBC it is possible to make a predictable change to plaintext block n, for instance flipping bit x, by flipping bit x in ciphertext block n−1. There is a side-effect, though, of completely garbling plaintext block n−1. Whether an attacker can use this for nefarious purposes depends on the exact application. With CBC done on the outside, an attacker can still do the same

attack. The fact that the encryption scheme involves triple use of DES does not change the effects of CBC. An attacker that flips bit x in ciphertext block n−1 will completely and unpredictably (to the attacker) garble plaintext block n−1. However, plaintext block n will have bit x flipped. And all plaintext blocks other than n−1 and n will be unaffected. 108 MODES OF OPERATION 4.5 With CBC done on the inside, any change to ciphertext block n completely and unpredictably garbles all plaintext blocks from n to the end of the message. This makes CBC done on the inside more secure, and perhaps would therefore have been a better choice. However, sometimes people would prefer if garbling of a ciphertext block did not garble the entire rest of the message. They'd prefer that the encryption scheme be self-synchronizing, which means that after some small number of garbled blocks, the plaintext will start decrypting properly again. There are also subtle security flaws with CBC on the inside if the attacker can supply chosen plaintext and IV and examine the output. Another advantage of CBC on the inside is performance. With CBC on the inside it is possible to use three times as much hardware and pipeline the encryptions so that it is as fast as single encryption. With CBC on the outside, this is not possible. One reason that people choose CBC on the outside despite its disadvantages is that EDE encryption can be considered a new secret key block encryption scheme that uses a 112-bit key. This can then be used with any of the chaining methods (OFB, ECB, CFB, CTR, as well as CBC). 4.5 HOMEWORK 1. What pseudo-random block stream is generated by 64-bit OFB with a weak DES key? 2. The pseudo-random stream of blocks generated by 64-bit OFB must eventually repeat (since at most 264 different blocks can be generated). Will K{IV} necessarily be the first block to be repeated? 3. Let's assume you do DES double encryption by encrypting with K1 and doing DES in decrypt mode with K2. Does the same attack work as with double encryption with K1 and K2? If not, how could it be made to work? 4. What is a practical method for finding a triple of keys that maps a given plaintext to a given ciphertext using EDE? Hint: It is like the meet-in-the-middle attack of §4.4.1.2 Encrypting Twice with Two Keys. 5. Let's assume that someone does triple encryption by using EEE with CBC on the inside. Suppose an attacker modifies bit x of ciphertext block n. How does this affect the decrypted plaintext? 6. Consider the following alternative method of encrypting a message. To encrypt a message, use the algorithm for doing a CBC decrypt. To decrypt a message, use the algorithm for doing 4.5 HOMEWORK 109 a CBC encrypt. Would this work? What are the security implications of this, if any, as contrasted with the "normal" CBC? 7. Give the first 16 values of the permutation formed by doing the DES initial permutation twice (as happens when ED is performed). This page is intentionally left blank 111 5 HASHES AND MESSAGE DIGESTS 5.1 INTRODUCTION Random numbers should not be generated with a method chosen at random. —Donald Knuth A hash (also known as a message digest) is a one-way function. It is considered a function because it takes an input message and produces an output. It is considered one-way because it's not practical to figure out what input corresponds to a given output. For a message digest function to be considered cryptographically secure, it must be computationally infeasible to find a message that has a given prespecified message digest, and it similarly should be impossible to find two messages that have the same message digest. Also (which follows from the previous properties), given a message it should be impossible to find a different message with the same message digest. We will use the terms hash and message digest interchangeably. The NIST message digest function is called SHA-1, which stands for secure hash algorithm, whereas the MD in the MD2, MD4, and MD5 algorithms stands for message digest. All of the digest/hash algorithms do basically the same thing, which is to take an arbitrary-length message and wind up with a fixed-length quantity. There's an intuitive notion of randomness that is important to the understanding of message digest functions. We are not trying to create a math book, so our description will be mathematically

imprecise. We are merely trying to capture the intuition. We explained before that with secret key cryptography, it is desirable for the mapping from input to output to appear randomly chosen. In other words, it should look (to someone who does not know the secret key) like someone flipped coins to determine, for each possible input, what the output should be. Examples of what this means are: • If 1000 inputs are selected at random, any particular bit in the 1000 resulting outputs should be on about half the time. 112 HASHES AND MESSAGE DIGESTS 5.1 • Each output should have, with high probability, about half the bits on. • Any two outputs should be completely uncorrelated, no matter how similar the inputs are. So for instance, two inputs that differ by only one bit should have outputs that look like completely independently chosen random numbers. About half the bits in the two outputs should differ. This sort of randomness is important for message digests as well. It is true that someone who knows the message and the digest function can calculate the output, and therefore the output is certainly provably not generated by flipping coins. But ignoring that, the output should look random. It should not be possible, other than by computing the message digest, to predict any portion of the output. And in particular it should be true that, for any subset of bits in the message digest, the only way of obtaining two messages with the same value in those bits would be to try messages at random and compute the message digest of each until two happened to have the same value. Given this property, a secure message digest function with n bits should be derivable from a message digest function with more than n bits merely by taking any particular subset of n bits from the larger message digest. There certainly will be many messages that yield the same message digest, because a message can be of arbitrary length and the message digest will be some fixed length, for instance 128 bits. For instance, for 1000-bit messages and a 128-bit message digest, there are on the average 2872 messages that map to any one particular message digest. So certainly, by trying lots of messages, one would eventually find two that mapped to the same message digest. The problem is that “lots” is so many that it is essentially impossible. Assuming a good 128-bit message digest function, it would take trying approximately 2128 possible messages before one would find a message that mapped to a particular message digest, or approximately 264 messages before finding two that had the same digest (see The Birthday Problem, below). An example use of a message digest is to fingerprint a program or document to detect modification of it. If you know the message digest of the program (and store the message digest securely so that it can’t be modified, and compute the message digest of the program before running it, and check to make sure that it matches the stored value), then nobody will be able to modify the program without being detected, because they will not be able to find a different program with the same message digest. How many bits does the output of a message digest function have to be in order to prevent someone from being able to find two messages with the same message digest? Well, if the message digest has m bits, then it would take only about 2m/2 messages, chosen at random, before one would find two with the same value. So if there were a 64-bit message digest function, it would only take searching 232 messages before one could find two with the same value, and it is feasible to search 232 messages. That is why message digest functions have outputs of at least 128 bits, because it is not considered feasible to search 264 messages given the current state of the art. 5.1 INTRODUCTION 113 If cryptographers were happy with message digest functions that merely made it infeasible to find a message with a particular prespecified digest (rather than being worried about being able to find any pair with the same digest), then a 64-bit digest would suffice. Why are we worried about someone being able to find any two messages with the same message digest, instead of only worrying about finding a message with a particular prespecified message digest? In most applications, to subvert a system an attacker has to find a misleading message whose message digest matches that of a

pre-existing message. However, there are cases where being able to find two messages with the same message digest is a threat to the security of the system. Suppose Alice wants to fire Fred, and asks her diabolical secretary, Bob, who happens to be Fred's friend, to compose a letter explaining that Fred should be fired, and why. After Bob writes the letter, Alice will read it, compute a message digest, and cryptographically sign the message digest using her private key. Bob would like to instead write a letter saying that Fred is wonderful and his salary ought to be doubled. However, Bob cannot generate a message digest signed with Alice's key. If he can find two messages with the same message digest, one that Alice will look at and agree to sign because it captures what she'd like it to say, and one that says what Bob would like it to say, then Bob can substitute his own message after Alice generates the signed message digest. Suppose the message digest function has only 64 bits, and is a good message digest function in the sense of its output looking random. Then the only way to find two messages with the same message digest would be by trying enough messages so that by the birthday problem two would have the same digest. If Bob started by writing a letter that Alice would approve of, found the message digest of that, and then attempted to find a different message with that message digest, he'd have to try 264 The Birthday Problem If there are 23 or more people in a room, the odds are better than 50% that two of them will have the same birthday. Analyzing this parlor trick can give us some insight into cryptography. We'll assume that a birthday is basically an unpredictable function taking a human to one of 365 values (yeah yeah, 366 for you nerds). Let's do this in a slightly more general way. Let's assume n inputs (which would be humans in the birthday example) and k possible outputs, and an unpredictable mapping from input to output. With n inputs, there are n(n−1)/2 pairs of inputs. For each pair there's a probability of 1/k of both inputs producing the same output value, so you'll need about k/2 pairs in order for the probability to be about 50% that you'll find a matching pair. That means that if n is greater than , there's a good chance of finding a matching pair. k 114 HASHES AND MESSAGE DIGESTS 5.1 different messages. However, suppose he had a way of generating lots of messages of each type (type 1—those that Alice would be willing to sign; type 2—those that Bob would like to send). Then by the birthday problem he'd only have to try about 232 messages of each type before he found two that matched. (See Homework Problem 3.) How can Bob possibly generate that many letters, especially since they'd all have to make sense to a human? Well, suppose there are 2 choices of wording in each of 32 places in the letter. Then there are 232 possible messages he can generate. For example: Type 1 message I am writing {this memo | } to {demand | request | inform you} that {Fred | Mr. Fred Jones} {must | } be {fired | terminated} {at once | immediately}. As the {July 11 | 11 July} {memo | memorandum} {from | issued by} {personnel | human resources} states, to meet {our | the corporate} {quarterly | third quarter} budget {targets | goals}, {we must eliminate all discretionary spending | all discretionary spending must be eliminated}. {Despite | Ignoring} that {memo | memorandum | order}, Fred {ordered | purchased} {PostIts | nonessential supplies} in a flagrant disregard for the company's {budgetary crisis | current financial difficulties}. Type 2 message I am writing {this letter | this memo | this memorandum | } to {officially | } commend Fred {Jones | } for his {courage and independent thinking | independent thinking and courage}. {He | Fred} {clearly | } understands {the need | how} to get {the | his} job {done | accomplished} {at all costs | by whatever means necessary}, and {knows | can see} when to ignore bureaucratic {nonsense | impediments}. I {am hereby recommending | hereby recommend} {him | Fred} for {promotion | immediate advancement} and {further | } recommend a {hefty | large} {salary | compensation} increase. There are enough computer-generatable variants of the two letters that Bob can compute message digests on the various variants until he finds a match. It is within computational feasibility to generate and test on the order of 232 messages, whereas it would not be feasible to deal with 264 messages. As

we will see in Chapter 6 Public Key Algorithms, the math behind RSA and other public key cryptographic algorithms is quite understandable. In contrast, message digest functions are like alchemy. It's a bunch of steps that each mangle the message more and more. The functions sound like a bunch of people got together and each person had some idea for mangling the message further. "And now let's swap every bit with the complement of the bit 11 bits to the right!" "How about let's multiply every set of 12 adjacent bits by the constant 384729?" "How about we throw in I(newt)?" And every suggestion is adopted by the group. It is possible to follow all the steps, but the 5.2 NIFTY THINGS TO DO WITH A HASH 115 incomprehensible part is why this particular shuffling of the message is done, and why it couldn't be simpler and yet just as secure. A plausible way of constructing a message digest function is to combine lots of perverse operations into a potential message digest function, and then play with it. If any patterns are detected in the output the function is rejected, or perhaps more perverse operations are folded into it. Ideally, the message digest function should be easy to compute. One wonders what the "minimal" secure message digest function might be. It is safer for a function to be overkill, in the sense of shuffling beyond what it necessary, but then it is harder to compute than necessary. The designers would rather waste computation than discover later that the function was not secure. Just as with secret key algorithms, the digest algorithms tend to be computed in rounds. It is likely that the designers find the smallest number of rounds necessary before the output passes various randomness tests, and then do a few more just to be safe. There is an interesting bit of folklore in the choice of constants in cryptographic algorithms. Often a "random number" is needed. If the designer of the algorithm chooses a specific number without explanation as to where that number came from, people are suspicious that there might be some hidden flaw that the designer of the algorithm wishes to exploit, and will be able to because the designer knows some hidden properties of that choice of number. Therefore, often the algorithm designers specify how they chose a particular random number. It is often done based on the digits of π or some other irrational number. There was also a book published in the 1930s of random numbers generated from a mechanically random source. People have used some of the numbers in that as well on the theory that cryptography was not well understood then so it is inconceivable that someone could have planted special numbers. At the end of this chapter we'll describe the major hash functions which have been proposed as standards. Ron Rivest designed MD2, MD4, and MD5. These are all documented in RFCs (1319, 1320, and 1321). In MD2 all operations are done with octets (the microprocessors at the time operated on octets), whereas the other standards operate on 32-bit words (today's microprocessors efficiently compute with 32-bit quantities). In fact, now that there are 64-bit microprocessors, it's quite possible a new MD algorithm optimized for that architecture will be developed. 5.2 NIFTY THINGS TO DO WITH A HASH Before we look at the details of several popular hash algorithms, let's look at some interesting uses of hash algorithms. Surprisingly, if there is a shared secret, the hash algorithms can be used in all the ways that secret cryptography is used. It is a little confusing calling the schemes in the next few sections "message digest" schemes, since they do involve a shared secret. By our definition in §2.3 116 HASHES AND MESSAGE DIGESTS 5.2 Types of Cryptographic Functions in which we said something that had a single shared secret was a secret key algorithm, these might be considered secret key algorithms. Never take definitions too seriously. The significant difference between a secret key algorithm and a message digest algorithm is that a secret key algorithm is designed to be reversible and a message digest algorithm is designed to be impossible to reverse. In this section we'll use MD as a "generic" message digest (cryptographic hash) algorithm. Why are there so many message digest functions? Surprisingly, the drive for message digest algorithms started with public key cryptography. RSA was invented,

which made possible digital signatures on messages, but computing a signature on a long message with RSA was sufficiently slow that RSA would not have been practical by itself. A cryptographically secure message digest function with high performance would make RSA much more useful. Instead of having to compute a signature over a whole long message, the message could be compressed into a small size by first performing a message digest and then computing the RSA signature on the digest. So MD and MD2 were created. MD was proprietary and never published. It was used in some of RSADSI's secure mail products. MD2 is documented in RFC 1319. Then Ralph Merkle of Xerox developed a message digest algorithm called SNEFRU [MERK90] that was several times faster than MD2. This prodded Ron Rivest into developing MD4 (RFC 1320), a digest algorithm that took advantage of the fact that newer processors could do 32-bit operations, and was therefore able to be even faster than SNEFRU. Then SNEFRU was broken [BIHA92] (the cryptographic community considered it broken because someone was able to find two messages with the same SNEFRU digest). Independently, [DENB92] found weaknesses in a version of MD4 with two rounds instead of three. This did not officially break MD4, but it made Ron Rivest sufficiently nervous that he decided to strengthen it, and create MD5 (RFC 1321), which is a little slower than MD4. NIST subsequently proposed SHA, which is very similar to MD5, but even more strengthened, and also a little slower. Probably after discovering a never published flaw in the SHA proposal, NIST revised it at the twelfth hour in an effort to make it more secure, and called the revised version SHA-1 (see §5.6.3 SHA-1 Operation on a 512-bit Block). MD2 and MD4 were subsequently broken (in the sense that collisions were found), though they remain secure for most uses. At the time of this writing, NIST was working on a new hash function (probably to be named SHA-2) to increase the number of bits of output to 256 in order to have security comparable to the AES encryption algorithm. Yes, Virginia, there was an MD3, but it was superseded by MD4 before it was ever published or used. 5.2.1 NIFTY THINGS TO DO WITH A HASH 117 5.2.1 Authentication In §2.4.4 Authentication we discussed how to use a secret key algorithm for authentication. A challenge is transmitted, and the other side has to encrypt the challenge with the shared key. Imagine a world without secret key cryptography, but with cryptographic hash functions. So we can't use an algorithm like DES in the above example. This is not an entirely theoretical concern. Export controls may treat secret key algorithms more harshly than digest algorithms even if they are only used to compute MACs, especially when source code is provided. Could we use a message digest function in some way to accomplish the same thing? Bob and Alice will still need to share a secret. Message digest algorithms aren't reversible, so it can't work quite the same way. In the above example, in which secret key cryptography is used for authentication, Bob encrypts something, and Alice decrypts it to make sure Bob encrypted the quantity properly. A hash function will do pretty much the same thing. Alice still sends a challenge. Bob then concatenates the secret he shares with Alice with the challenge, takes a message digest of that, and transmits that message digest. Alice can't "decrypt" the result. However, she can do the same computation, and check that the result matches what Bob sends. 5.2.2 Computing a MAC with a Hash In §4.3 Generating MACs we described how to compute a MAC (message authentication code) with a secret key algorithm. Again, let's assume that for some reason no secret key algorithms are available. Can we still compute a MAC, using a hash function instead of something like DES? The obvious thought is that MD(m) is a MAC for message m. But it isn't. Anyone can compute MD(m). The point of the MAC is to send something that only someone knowing the secret can compute (and verify). For instance, if Alice and Bob share a secret, then Alice can send m, plus Alice Bob rA rA encrypted with KAB rB rB encrypted with KAB Alice Bob rA MD(KAB|rA) rB MD(KAB|rB) 118 HASHES AND MESSAGE DIGESTS 5.2.2 MAC, and since nobody except Alice and Bob can compute a MAC with their

shared key, nobody but Alice or Bob would be able to send a message to Bob with an appropriate MAC. If we just simply used MD, then anyone can send any message m' together with MD(m'). So we do roughly the same trick for the MAC as we did for authentication. We concatenate a shared secret KAB with the message m, and use MD(KAB|m) as the MAC. This scheme almost works, except for some idiosyncracies of most of the popular message digest algorithms, which would allow an attacker to be able to compute a MAC of a longer message beginning with m, given message m and the correct MAC for m. Assume MD is one of MD4, MD5, or SHA-1. The way these algorithms work is that the message is padded to a multiple of 512 bits with a pad that includes the message length. The padded message is then digested from left to right in 512-bit chunks. In order to compute the message digest through chunk n, all that you need to know is the message digest through chunk n−1, plus the value of chunk n of the padded message. Let's assume Carol would like to send a different message to Bob, and have it look like it came from Alice. Let's say that Carol doesn't care what's in the message. She only cares that the end of the message says P.S. Give Carol a promotion and triple her salary. Alice has transmitted some message m, and MD(KAB|m). Carol can see both of those quantities. She concatenates the padding and then whatever she likes to the end of m, and initializes the message digest computation with MD(KAB|m). She does not need to know the shared secret in order to compute the MAC. How can we avoid this flaw? Lots of techniques have been proposed, all entirely effective as far as anyone can tell. But people came up with "improvements", each one a little more complex than the one before, with the apparent winner being HMAC. Some proposals with no known weaknesses are: • Put the secret at the end of the message instead of at the beginning. This will work. This method can be criticized for an extremely unlikely security flaw. The complaint is that if the MD algorithm were weak, and it was therefore possible to find two messages with the same digest, then those two messages would also have the same MAC. • Use only half the bits of the message digest as the MAC. For instance, take the low-order 64 bits of the MD5 digest. This gives an attacker no information with which to continue the message digest (well, the attacker has a 1 in 264 chance of guessing the rest of the message digest correctly—we assume you're not going to worry about that risk). Having only 64 bits of MAC (rather than using all 128 bits of the MD) is not any less secure, since there is no way that an attacker can generate messages and test the resulting MAC. Without knowing the secret, there is no way for the attacker to calculate the MAC. The best that can be done is to generate a random 64-bit MAC for the message you'd like to send and hope that you'll be really really lucky. 5.2.3 NIFTY THINGS TO DO WITH A HASH 119 • Concatenate the secret to both the front and the back of the message. That way you get the collision resistance of putting it in front and the protection from appending that comes from putting it in back. HMAC concatenates the secret to the front of the message, digests the combination, then concatenates the secret to the front of the digest, and digests the combination again. The actual construction is a little more complicated than this, and is described in section §5.7 HMAC. HMAC has lower performance than the other alternatives because it does a second digest. But the second digest is only computed over the secret and a digest, so it does not add much cost to large messages. In the worst case, if the message concatenated with the key fit into a single (512-bit) block, HMAC would be four times as expensive as one of the other alternatives described above. However, if many small messages are to be HMAC'd with the same key, it is possible to reuse the part of the computation that digests the key, so that HMAC would only be twice as slow. With a large enough message, HMAC's performance is only negligibly worse. We call any hash combining the secret key and the data a keyed hash. 5.2.3 Encryption with a Message Digest "Encryption with a message digest algorithm is easy!" you say. "But let me see you do decryption!" Message digest algorithms are not reversible, so the

trick is to design a scheme in which both encryption and decryption run the message digest algorithm in the forward direction. The schemes we'll describe are reminiscent of the chaining methods for a secret key algorithm (see §4.2 Encrypting a Large Message). 5.2.3.1 Generating a One-Time Pad Just as OFB (§4.2.3 Output Feedback Mode (OFB)) generates a pseudorandom bit stream which then encrypts a message by simply being ⊕ed with the message, we can use a message digest algorithm to generate a pseudorandom bit stream. Again, Alice and Bob need a shared secret, KAB. Alice wants to send Bob a message. She computes MD(KAB). That gives the first block of the bit stream, b1. Then she computes MD(KAB|b1) and uses that as b2, and in general bi is MD(KAB|bi−1). Alice and Bob can do this in advance, before the message is known. Then when Alice wishes to send the message, she ⊕s it with as much of the generated bit stream as necessary. Similarly, Bob decrypts the ciphertext by ⊕ing it with the bit stream he has calculated. It is not secure to use the same bit stream twice, so, as with OFB, Alice starts with an IV. The first block is then MD(KAB|IV). She must transmit the IV to Bob. Alice can generate the bit stream in advance of encrypting the message, but Bob cannot generate the bit stream until he sees the IV. 120 HASHES AND MESSAGE DIGESTS 5.2.3.2 5.2.3.2 Mixing In the Plaintext One-time pad schemes have the problem that if you are able to guess the plaintext, you can ⊕ the guessed text with the ciphertext, and then ⊕ any message you like. This is not too much of a problem. We just need to recognize that a one-time pad scheme gives privacy only, and integrity must be gained through a scheme such as using a MAC. However, in a scheme similar to CFB (§4.2.4 Cipher Feedback Mode (CFB)), we can mix the plaintext into the bit stream generation. For instance, break the message into MD-length chunks p1, p2,.... We'll call the ciphertext blocks c1, c2,.... And we'll need intermediate values b1, b2,... from which we'll compute each ciphertext block. Decryption is straightforward. We leave it as Homework Problem 18. 5.2.4 Using Secret Key for a Hash In case the previous sections make the secret key algorithms nervous about job security since they can be replaced by hash algorithms, we'll show that a hash algorithm can be replaced by a secret key algorithm. What we want to generate is a function with the properties of a hash algorithm. It should not require a secret. It should be publishable. It should be noninvertible. 5.2.4.1 UNIX Password Hash UNIX uses a secret key algorithm to compute the hash of a password, which it then stores. It never has to reverse the hash to obtain a password. Instead, when the user types a password, UNIX uses the same algorithm to hash the typed quantity and compares the result with the stored quantity. The hashing algorithm first converts the password into a secret key. This key is then used, with a DES-like algorithm, to encrypt the number 0. The method of turning a text string into a secret key is simply to pack the 7-bit ASCII associated with each of the first 8 characters of the password into a 56-bit quantity into which DES parity is inserted. (UNIX passwords can be longer than 8 characters, but the remaining octets are ignored.) b1 = MD(KAB|IV) c1 = p1 ⊕ b1 b2 = MD(KAB|c1) c2 = p2 ⊕ b2 ... ... bi = MD(KAB|ci−1) ci = pi ⊕ bi ... ... 5.2.4.2 NIFTY THINGS TO DO WITH A HASH 121 A 12-bit random number, known as salt, is stored with the hashed password. For an explanation of why salt is useful, see §8.3 Off-Line Password Guessing. A modified DES is used instead of standard DES to prevent hardware accelerators designed for DES from being used to reverse the password hash. The salt is used to modify the DES data expansion algorithm. The value of the salt determines which bits are duplicated when expanding R from 32 to 48 bits (see §3.3.5 The Mangler Function). To summarize, each time a password is set, a 12-bit number is generated. The password is converted into a secret key. The 12-bit number is used to define a modified DES algorithm. The modified DES algorithm is used with the secret key as input to encrypt the constant 0. The result is stored along with the 12-bit number as the user's hashed password. 5.2.4.2 Hashing Large Messages The UNIX password hash is a method of doing a message digest of a very short message (maximum length is the

length of the secret key). Here's a method of converting a secret key algorithm into a message digest algorithm for arbitrary messages (see Figure 5-1). A secret key algorithm has a key length, say k bits. It has a message block length, say b bits. In the case of DES, k = 56 and b = 64. In the case of IDEA, k = 128 and b = 64. Divide the message into k-bit chunks m1, m2,.... Use the first block of the message as a key to encrypt a constant. The result is a b-bit quantity. Use the second k-bit chunk of the message to encrypt the b-bit quantity to get a new b-bit quantity. Keep doing this until you run out of k-bit blocks of the message. Use the final b-bit result as the message digest. There's a serious problem with this, which is that the typical message block length b is 64 bits, which is too short to use as a message digest. To obtain two messages with the same message digest using this technique (and remembering the birthday problem), we'd only have to try about 232 messages before finding two that had the same digest. And furthermore, if we want to find a constant m1 key encrypt m2 key encrypt ... ... message digest Figure 5-1. Message Digest Using Secret Key Cryptography 122 HASHES AND MESSAGE DIGESTS 5.3 message with a particular message digest, a technique similar to the one in §4.4.1.2 Encrypting Twice with Two Keys could find a message with a particular 64-bit message digest in about 233 iterations. A technique that works better (and in particular makes it a workfactor of 263 to find a message matching a given hash) is to ⊕ the input to each round with the output as shown below: One possible technique for generating 128 bits of message digest is to generate two 64-bit quantities using techniques that are similar to compute but designed to produce different values. The first 64-bit quantity might be generated as we just described—the message is broken into key-length chunks b1,b2,... and the chunks are used for encryption in that order. The second 64-bit quantity is generated by using the chunks in reverse order. That technique has a flaw (see Homework Problem 4), and it's a little inconvenient to do two passes on the message. A better alternative is to process the message twice in the forward direction, and just start with two different constants. 5.3 MD2 MD2 takes a message equal to an arbitrary number of octets and produces a 128-bit message digest. It cannot handle a message that is not an integral number of octets, though it would be simple to constant m1 key encrypt ⊕ m2 key encrypt ⊕ ... ... message digest Figure 5-2. Improved Message Digest Using Secret Key Cryptography 5.3.1 MD2 123 modify MD2 (see Homework Problem 5), or to have a convention for bit-padding a message before feeding it to MD2. The basic idea behind MD2 is as follows: 1. The input to MD2 is a message whose length is an arbitrary number of octets. 2. The message is padded, according to specified conventions, to be a multiple of 16 octets. 3. A 16-octet quantity, which MD2 calls a checksum, is appended to the end. This checksum is a strange function of the padded message defined specifically for MD2. 4. Final pass—The message is processed, 16 octets at a time, each time producing an intermediate result for the message digest. Each intermediate value of the message digest depends on the previous intermediate value and the value of the 16 octets of the message being processed. Note that we describe the algorithm as if the message is processed three times, first to pad it, next to compute the checksum, and then to compute the actual message digest. However, it is possible to digest a message in a single pass. This is important for machines with limited memory. 5.3.1 MD2 Padding The padding function (see Figure 5-3) is very simple. There must always be padding, even if the message starts out being a multiple of 16 octets. If the message starts out being a multiple of 16 octets, 16 octets of padding are added. Otherwise, the number of octets (1–15) necessary to make the message a multiple of 16 octets is added. Each pad octet specifies the number of octets of padding that was added. 5.3.2 MD2 Checksum Computation The checksum is a 16-octet quantity. It is almost like a message digest, but it is not cryptographically secure by itself. The checksum is appended to the message, and then MD2 processes the concatenated quantity to obtain the actual message digest. original message r

octets (1 ≤ r ≤ 16) each containing r multiple of 16 octets Figure 5-3. MD2 Padded Message 124 HASHES AND MESSAGE DIGESTS 5.3.2 The checksum starts out initialized to 0. Because of the padding mentioned in the previous section, the message is a multiple of 16 octets, say k ×16 octets. The checksum calculation processes the padded message an octet at a time, so the calculation requires k ×16 steps. Each step looks at one octet of the message and updates one octet of the checksum. After octet 15 of the checksum is updated, the next step starts again on octet 0 of the checksum. Therefore, each octet of the checksum will have been updated k times by the time the checksum computation terminates. At step n, octet (n mod 16) of the checksum is computed, but for simplicity we'll call it octet n of the checksum. Octet n of the checksum depends on octet n of the message, octet (n−1) of the checksum, and the previous value of octet n of the checksum (what it was updated to be in step n−16). (Think of the checksum as "wrapping around", so if n = 0, octet n−1 is octet 15.) First, octet n of the message and octet (n−1) of the checksum are ⊕'d together. This produces an octet having a value between 0 and 255. Then a substitution is done and the result is ⊕'d into the previous value of octet n of the checksum. The substitution is specified in Figure 5-5. The first entry is 41, indicating that the value 0 is mapped to 41. The next is 46, indicating that the value 1 is mapped to 46. The designers of MD2, anxious to display that they had designed no sneaky trapdoors into MD2 by having carefully chosen a particular mapping function, specified that they chose the mapping based on the digits of π. One way that they could base a substitution on the digits of π was to look at the binary representation of π an octet at a time, using the value of that octet as the next substitution value, unless it was already used, in which case they skipped it and went on to the next padded message nth octet ⊕ π substitution ⊕ 16-octet checksum (n−1 mod 16)th octet (n mod 16)th octet Figure 5-4. MD2 Checksum Calculation 5.3.3 MD2 125 octet. We checked, and that isn't how they did it. We're willing to just take their word for it that somehow the numbers are based on the digits of π. We wonder whether anyone has ever bothered to verify their claim. We certainly hope none of you waste as much time as we did trying to figure out how the digits of π relate to the substitution. We could have asked the designers, but that seemed like cheating. 5.3.3 MD2 Final Pass The final pass of the message digest computation is somewhat similar to the checksum computation. The message with padding and checksum appended is processed in chunks of 16 octets. Each time a new 16-octet chunk of the message is taken, a 48-octet quantity is constructed consisting of the (16-octet) current value of the message digest, the chunk of the message, and the ⊕ of those two 16-octet quantities. The 48-octet quantity is massaged octet by octet. After 18 passes over the 48-octet quantity, the first 16 octets of the 48-octet quantity are used as the next value of the message digest. The message digest is initialized to 16 octets of 0. At each stage, the next 16 octets of the message are taken to be processed. A 48-octet quantity is formed by appending the message octets to the current message digest, then appending the ⊕ of these two 16-octet quantities. 41 46 67 201 162 216 124 1 61 54 84 161 236 240 6 19 98 167 5 243 192 199 115 140 152 147 43 217 188 76 130 202 30 155 87 60 253 212 224 22 103 66 111 24 138 23 229 18 190 78 196 214 218 158 222 73 160 251 245 142 187 47 238 122 169 104 121 145 21 178 7 63 148 194 16 137 11 34 95 33 128 127 93 154 90 144 50 39 53 62 204 231 191 247 151 3 255 25 48 179 72 165 181 209 215 94 146 42 172 86 170 198 79 184 56 210 150 164 125 182 118 252 107 226 156 116 4 241 69 157 112 89 100 113 135 32 134 91 207 101 230 45 168 2 27 96 37 173 174 176 185 246 28 70 97 105 52 64 126 15 85 71 163 35 221 81 175 58 195 92 249 206 186 197 234 38 44 83 13 110 133 40 132 9 211 223 205 244 65 129 77 82 106 220 55 200 108 193 171 250 36 225 123 8 12 189 177 74 120 136 149 139 227 99 232 109 233 203 213 254 59 0 29 57 242 239 183 14 102 88 208 228 166 119 114 248 235 117 75 10 49 68 80 180 143 237 31 26 219 153 141 51 159 17 131 20 Figure 5-5. MD2 π Substitution Table 126 HASHES AND MESSAGE DIGESTS 5.3.3 18 passes are made over the 48-

octet quantity, and during a pass each step processes one octet, which means that 18×48 steps are made. It is necessary to know which pass is being made, since the pass number is used in the computation. Both passes and steps are numbered starting at zero. A phantom octet −1 appears before the first octet (octet 0) of the 48-octet quantity. At the beginning of pass 0, octet −1 is set to 0. In step n of each pass we take octet n−1 and run it through the same substitution used for the checksum, then ⊕ the result into octet n. After step 47 of each pass, there is a step 48 which sets octet −1 to the mod 256 sum of octet 47 and the pass number. When we have completed pass 17, the first 16 octets of the 48-octet quantity are used as the value of the message digest for the next stage, and the next 16 octets of the message are processed. (So in pass 17, steps 16 through 48 are useless.) After the entire message is processed, the 16 octets that come out of the last stage is the message digest value. The calculation requires only 16 octets (for the checksum) plus 48 octets of volatile memory plus a few indices and makes a single pass over the data. This is ideal for computation in a smart card (see §8.8 Authentication Tokens). padded message with appended 16-octet checksum 16-octet block initial value = 0 MD intermediate message block 0 ⊕ octet n−1 → ← octet n octet 47 → + pass # (0–17) π substitution ⊕ for n from 0 thru 47 for next message block discarded Final MD2 after checksum processed Figure 5-6. MD2 Final Pass 5.4 MD4 127 5.4 MD4 MD4 was designed to be 32-bit-word-oriented so that it can be computed faster on 32-bit CPUs than an octet-oriented scheme like MD2. Also, MD2 requires the message to be an integral number of octets. MD4 can handle messages with an arbitrary number of bits. Like MD2 it can be computed in a single pass over the data, though MD4 needs more intermediate state. 5.4.1 MD4 Message Padding The message to be fed into the message digest computation must be a multiple of 512 bits (sixteen 32-bit words). The original message is padded by adding a 1 bit, followed by enough 0 bits to leave the message 64 bits less than a multiple of 512 bits. Then a 64-bit quantity representing the number of bits in the unpadded message, mod 264, is appended to the message. The bit order within octets is most significant to least significant, the octet order is least significant to most significant. 5.4.2 Overview of MD4 Message Digest Computation The message digest to be computed is a 128-bit quantity (four 32-bit words). The message is processed in 512-bit (sixteen 32-bit words) blocks. The message digest is initialized to a fixed value, and then each stage of the message digest computation takes the current value of the message digest and modifies it using the next block of the message. The function that takes 512 bits of the message and digests it with the previous 128-bit output is known as the compression function. The final result is the message digest for the entire message. Each stage makes three passes over the message block. Each pass has a slightly different method of mangling the message digest. At the end of the stage, each word of the mangled message digest is added to its pre-stage value to produce the post-stage value (which becomes the pre-stage value for the next stage). Therefore, the current value of the message digest must be saved at the beginning of the stage so that it can be added in at the end of the stage. 1–512 bits 64 bits original message 1000…000 original length in bits multiple of 512 bits Figure 5-7. Padding for MD4, MD5, SHA-1 128 HASHES AND MESSAGE DIGESTS 5.4.2 Each stage starts with a 16-word message block and a 4-word message digest value. The message words are called m0, m1, m2,…m15. The message digest words are called d0, d1, d2, d3. Before the first stage the message digest is initialized to d0 = 6745230116, d1 = efcdab8916, d2 = 98badcfe16, and d3 = 1032547616, equivalent to the octet string (written as a concatenation of hex-encoded octets) 01|23|45|67|89|ab|cd|ef|fe|dc|ba|98|76|54|32|10. Each pass modifies d0, d1, d2, d3 using m0, m1, m2,…m15. We will describe what happens in each pass separately. The computations we are about to describe use the following operations: • Îx° is the floor of the number x, i.e., the greatest integer not greater than x. • ∼x is the bitwise complement of the 32-bit quantity x. • x∧y

is the bitwise and of the 32-bit quantities x and y. • x∨y is the bitwise or of the two 32-bit quantities x and y. • x⊕y is the bitwise exclusive or of the 32-bit quantities x and y. • x+y is the binary sum of the two 32-bit quantities x and y, with the carry out of the high order bit discarded. • x↲y is the 32-bit quantity produced by taking the 32 bits of x and shifting them one position left y times, each time taking the bit shifted off the left end and placing it as the rightmost bit. This operation is known as a left rotate. constant padded message ... digest 512 bits digest 512 bits . . . . . digest 512 bits Message Digest Figure 5-8. Overview of MD4, MD5, SHA-1 5.4.3 MD4 129 5.4.3 MD4 Message Digest Pass 1 A function F(x,y,z) is defined as (x∧y)∨(∼x∧z). This function takes three 32-bit words x, y, and z, and produces an output 32-bit word. This function is sometimes known as the selection function, because if the nth bit of x is a 1 it selects the nth bit of y for the nth bit of the output. Otherwise (if the nth bit of x is a 0) it selects the nth bit of z for the nth bit of the output. A separate step is done for each of the 16 words of the message. For each integer i from 0 through 15, d(−i)∧3 = (d(−i)∧3 + F(d(1−i)∧3, d(2−i)∧3, d(3−i)∧3) + mi) ↲ S1(i∧3) where S1(i) = 3 + 4i, so the ↲s cycle over the values 3, 7, 11, 15. If you don't find the previous sentence intimidating, you may go on to the next section. However, if you are a mere mortal, we'll explain, but just this once. The other passes in MD4 and MD5 are extremely similar, and we'll assume you'll understand their description with no further explanation. The "∧3" that appears several times in the above equation means that only the bottom two bits are used (because we're doing a bitwise and with 112). So i∧3 cycles 0, 1, 2, 3, 0, 1, 2, 3, ... while (−i)∧3 cycles 0, 3, 2, 1, 0, 3, 2, 1,... and (1−i)∧3 cycles 1, 0, 3, 2, 1, 0, 3, 2,.... We can write out the first few steps of the pass as follows: d0 = (d0 + F(d1, d2, d3) + m0) ↲ 3 d3 = (d3 + F(d0, d1, d2) + m1) ↲ 7 d2 = (d2 + F(d3, d0, d1) + m2) ↲ 11 d1 = (d1 + F(d2, d3, d0) + m3) ↲ 15 d0 = (d0 + F(d1, d2, d3) + m4) ↲ 3 5.4.4 MD4 Message Digest Pass 2 A function G(x, y, z) is defined as (x∧y)∨(x∧z)∨(y∧z). This function is sometimes known as the majority function, because the nth bit of the output is a 1 iff at least two of the three input words' nth bits are a 1. As for pass 1, we'll write out the first few steps. Note that in pass 2 (and pass 3 as well), the words of the message are not processed in order. Also note that there's a strange constant thrown in. To show that the designers didn't purposely choose a diabolical value of the constant, the constant is based on the square root of 2. The constant is □□ = 5a82799916. A separate step is done for each of the 16 words of the message. For each integer i from 0 through 15, d(−i)∧3 = (d(−i)∧3 + G(d(1−i)∧3, d(2−i)∧3, d(3−i)∧3) + mX(i) + 5a82799916) ↲ S2(i∧3) 2 2 30 130 HASHES AND MESSAGE DIGESTS 5.4.5 where X(i) is the 4-bit number formed by exchanging the low order and high order pairs of bits in the 4-bit number i (so X(i) = 4i − 15 □i/4□), and S2(0) = 3, S2(1) = 5, S2(2) = 9, S2(3) = 13, so the ↲s cycle over the values 3, 5, 9, 13. We can write out the first few steps of the pass as follows: d0 = (d0 + G(d1, d2, d3) + m0 + 5a82799916) ↲ 3 d3 = (d3 + G(d0, d1, d2) + m4 + 5a82799916) ↲ 5 d2 = (d2 + G(d3, d0, d1) + m8 + 5a82799916) ↲ 9 d1 = (d1 + G(d2, d3, d0) + m12 + 5a82799916) ↲ 13 d0 = (d0 + G(d1, d2, d3) + m1 + 5a82799916) ↲ 3 5.4.5 MD4 Message Digest Pass 3 A function H(x,y,z) is defined as x⊕y⊕z. Pass 3 has a different strange constant based on the square root of 3. The constant is □□ = 6ed9eba116. A separate step is done for each of the 16 words of the message. For each integer i from 0 through 15, d(−i)∧3 = (d(−i)∧3 + H(d(1−i)∧3, d(2−i)∧3, d(3−i)∧3) + mR(i) + 6ed9eba116) ↲ S3(i∧3) where R(i) is the 4-bit number formed by reversing the order of the bits in the 4-bit number i (so R(i) = 8i − 12 □i/2□ − 6 □i/4□ − 3 □i/8□), and S3(0) = 3, S3(1) = 9, S3(2) = 11, S3(3) = 15, so the ↲s cycle over the values 3, 9, 11, 15. We can write out the first few steps of the pass as follows: d0 = (d0 + H(d1, d2, d3) + m0 + 6ed9eba116) ↲ 3 d3 = (d3 + H(d0, d1, d2) + m8 + 6ed9eba116) ↲ 9 d2 = (d2 + H(d3, d0, d1) + m4 + 6ed9eba116) ↲ 11 d1 = (d1 + H(d2, d3, d0) + m12 + 6ed9eba116) ↲ 15 d0 = (d0 + H(d1, d2, d3) + m2 + 6ed9eba116) ↲ 3 5.5 MD5 MD5 was designed to be somewhat more "conservative" than MD4 in terms of being less concerned with speed and more concerned