with security. It is very similar to MD4. The major differences are: 1. MD4 makes three passes over each 16-octet chunk of the message. MD5 makes four passes over each 16-octet chunk. 2 3 30 5.5.1 MD5 131 2. The functions are slightly different, as are the number of bits in the shifts. 3. MD4 has one constant which is used for each message word in pass 2, and a different constant used for all of the 16 message words in pass 3. No constant is used in pass 1. MD5 uses a different constant for each message word on each pass. Since there are 4 passes, each of which deals with 16 message words, there are 64 32-bit constants used in MD5. We will call them T1 through T64. Ti is based on the sine function. Indeed, for any inquiring minds out there, Ti = ⌊232|sin i|⌋. The 64 values (in hex) are: 5.5.1 MD5 Message Padding The padding in MD5 is identical to the padding in MD4. 5.5.2 Overview of MD5 Message Digest Computation Like MD4, in MD5 the message is processed in 512-bit blocks (sixteen 32-bit words). (See Figure 5-8.) The message digest is a 128-bit quantity (four 32-bit words). Each stage consists of computing a function based on the 512-bit message chunk and the message digest to produce a new intermediate value for the message digest. The value of the message digest is the result of the output of the final block of the message. T1 = d76aa478 T17 = f61e2562 T33 = fffa3942 T49 = f4292244 T2 = e8c7b756 T18 = c040b340 T34 = 8771f681 T50 = 432aff97 T3 = 242070db T19 = 265e5a51 T35 = 6d9d6122 T51 = ab9423a7 T4 = c1bdceee T20 = e9b6c7aa T36 = fde5380c T52 = fc93a039 T5 = f57c0faf T21 = d62f105d T37 = a4beea44 T53 = 655b59c3 T6 = 4787c62a T22 = 02441453 T38 = 4bdecfa9 T54 = 8f0ccc92 T7 = a8304613 T23 = d8a1e681 T39 = f6bb4b60 T55 = ffeff47d T8 = fd469501 T24 = e7d3fbc8 T40 = bebfbc70 T56 = 85845dd1 T9 = 698098d8 T25 = 21e1cde6 T41 = 289b7ec6 T57 = 6fa87e4f T10 = 8b44f7af T26 = c33707d6 T42 = eaa127fa T58 = fe2ce6e0 T11 = ffff5bb1 T27 = f4d50d87 T43 = d4ef3085 T59 = a3014314 T12 = 895cd7be T28 = 455a14ed T44 = 04881d05 T60 = 4e0811a1 T13 = 6b901122 T29 = a9e3e905 T45 = d9d4d039 T61 = f7537e82 T14 = fd987193 T30 = fcefa3f8 T46 = e6db99e5 T62 = bd3af235 T15 = a679438e T31 = 676f02d9 T47 = 1fa27cf8 T63 = 2ad7d2bb T16 = 49b40821 T32 = 8d2a4c8a T48 = c4ac5665 T64 = eb86d391 132 HASHES AND MESSAGE DIGESTS 5.5.3 Each stage in MD5 takes four passes over the message block (as opposed to three for MD4). As with MD4, at the end of the stage, each word of the modified message digest is added to the corresponding pre-stage message digest value. And as in MD4, before the first stage the message digest is initialized to d0 = 6745230116, d1 = efcdab8916, d2 = 98badcfe16, and d3 = 1032547616. As with MD4, each pass modifies d0, d1, d2, d3 using m0 , m1, m2,...m15. We will describe what happens in each pass separately. 5.5.3 MD5 Message Digest Pass 1 As in MD4, F(x,y,z) is defined as the selection function (x∧y)∨(∼x∧z). 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(1−i)∧3 + (d(−i)∧3 + F(d(1−i)∧3, d(2−i)∧3, d(3−i)∧3) + mi + Ti+1) ↲ S1(i∧3) where S1(i) = 7 + 5i, so the ↲s cycle over the values 7, 12, 17, 22. This is a different S1 from that in MD4. We can write out the first few steps of the pass as follows: d0 = d1 + (d0 + F(d1, d2, d3) + m0 + T1) ↲ 7 d3 = d0 + (d3 + F(d0, d1, d2) + m1 + T2) ↲ 12 d2 = d3 + (d2 + F(d3, d0, d1) + m2 + T3) ↲ 17 d1 = d2 + (d1 + F(d2, d3, d0) + m3 + T4) ↲ 22 d0 = d1 + (d0 + F(d1, d2, d3) + m4 + T5) ↲ 7 5.5.4 MD5 Message Digest Pass 2 A function G(x,y,z) is defined as (x∧z)∨(y∧∼z). Whereas the function F was the same in MD5 as in MD4, the function G is different in MD5 than the G function in MD4. In fact, MD5's G is rather like F—the nth bit of z is used to select the nth bit in x or the nth bit in y. 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(1−i)∧3 + (d(−i)∧3 + G(d(1−i)∧3, d(2−i)∧3, d(3−i)∧3) + m(5i+1)∧15 + Ti+17) ↲ S2(i∧3) where S2(i) = i(i+7)/2 + 5, so the ↲s cycle over the values 5, 9, 14, 20. This is a different S2 from that in MD4. We can write out the first few steps of the pass as follows: d0 = d1 + (d0 + G(d1, d2, d3) + m1 + T17) ↲ 5 d3 = d0 + (d3 + G(d0, d1, d2) + m6 + T18) ↲ 9 d2 = d3 + (d2 + G(d3, d0, d1) + m11 + T19) ↲ 14 d1 = d2 + (d1 + G(d2, d3, d0) + m0 + T20) ↲ 20 5.5.5 MD5 133 d0 = d1 + (d0 + G(d1, d2, d3) + m5 + T21) ↲ 5 5.5.5 MD5

Message Digest Pass 3 A function H(x,y,z) is defined as x⊕y⊕z. This is the same H as in MD4. 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(1−i)∧3 + (d(−i)∧3 + H(d(1−i)∧3, d(2−i)∧3, d(3−i)∧3) + m(3i+5)∧15 + Ti+33) ↲ S3(i∧3) where S3(0) = 4, S3(1) = 11, S3(2) = 16, S3(3) = 23, so the ↲s cycle over the values 4, 11, 16, 23. This is a different S3 from MD4. We can write out the first few steps of the pass as follows: d0 = d1 + (d0 + H(d1, d2, d3) + m5 + T33) ↲ 4 d3 = d0 + (d3 + H(d0, d1, d2) + m8 + T34) ↲ 11 d2 = d3 + (d2 + H(d3, d0, d1) + m11 + T35) ↲ 16 d1 = d2 + (d1 + H(d2, d3, d0) + m14 + T36) ↲ 23 d0 = d1 + (d0 + H(d1, d2, d3) + m1 + T37) ↲ 4 5.5.6 MD5 Message Digest Pass 4 A function I(x,y,z) is defined as y⊕(x∨∼z). 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(1−i)∧3 + (d(−i)∧3 + I(d(1−i)∧3, d(2−i)∧3, d(3−i)∧3) + m(7i)∧15 + Ti+49) ↲ S4(i∧3) where S4(i) = (i+3)(i+4)/2, so the ↲s cycle over the values 6, 10, 15, 21. We can write out the first few steps of the pass as follows: d0 = d1 + (d0 + I(d1, d2, d3) + m0 + T49) ↲ 6 d3 = d0 + (d3 + I(d0, d1, d2) + m7 + T50) ↲ 10 d2 = d3 + (d2 + I(d3, d0, d1) + m14 + T51) ↲ 15 d1 = d2 + (d1 + I(d2, d3, d0) + m5 + T52) ↲ 21 d0 = d1 + (d0 + I(d1, d2, d3) + m12 + T53) ↲ 6 134 HASHES AND MESSAGE DIGESTS 5.6 5.6 SHA-1 SHA-1 (secure hash algorithm) was proposed by NIST as a message digest function. SHA-1 takes a message of length at most 264 bits and produces a 160-bit output. It is similar to the MD5 message digest function, but it is a little slower to execute and presumably more secure. MD4 made three passes over each block of data; MD5 made four; SHA-1 makes five. It also produces a 160-bit digest as opposed to the 128 of the MDs. 5.6.1 SHA-1 Message Padding SHA-1 pads messages in the same manner as MD4 and MD5, except that SHA-1 is not defined for a message that is longer than 264 bits. We don't consider that a problem. If there were such a message it would take several hundred years to transmit it at 10 Gigabits per second, and it would take even longer to compute the SHA-1 message digest for it at 1000 MIPS. As our grandmothers would surely have said, had the occasion arisen, If you can't say something in less that 264 bits, you shouldn't say it at all. 5.6.2 Overview of SHA-1 Message Digest Computation Just like MD4 and MD5, SHA-1 operates in stages (see Figure 5-8). Each stage mangles the pre-stage message digest by a sequence of operations based on the current message block. 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. The 160-bit message digest consists of five 32-bit words. Let's call them A, B, C, D, and E. Before the first stage they are set to A = 6745230116, B = efcdab8916, C = 98badcfe16, D = 1032547616, E = c3d2e1f016. After the last stage, the value of A|B|C|D|E is the message digest for the entire message. SHA-1 is extremely similar in structure to MD4 and MD5, even more so than their descriptions might indicate (see Homework Problem 7). 5.6.3 SHA-1 135 5.6.3 SHA-1 Operation on a 512-bit Block At the start of each stage, the 512-bit message block is used to create a 5×512-bit chunk (see Figure 5-9). The first 512 bits of the chunk consist of the message block. The rest gets filled in, a 32-bit word at a time, according to the bizarre rule that the nth word (starting from word 16, since words 0 through 15 consist of the 512-bit message block) is the ⊕ of words n−3, n−8, n−14, and n−16. In SHA-1, the ⊕ of words n−3, n−8, n−14, and n−16 is rotated left one bit before being stored as word n; this is the only modification from the original SHA. Now we have a buffer of eighty 32-bit words (5×512 bits). Let's call the eighty 32-bit words W0, W1, ...W79. Now, a little program: For t = 0 through 79, modify A, B, C, D, and E as follows: B = old A C = old B ↲ 30 D = old C E = old D "Hmm," you're thinking. "This isn't too hard to follow." Well, we haven't yet said what the new A is! Since the new A depends on the old A, B, C, D, and E, a programmer would first compute the new A into a temporary variable V, and then after computing the new values of B, C, D, and E, set the new A equal to V. For clarity we didn't

bother. In the following computation, everything to the right of the equal refers to the old values of A, B, C, D, and E: A = E + (A↲5) + Wt + Kt + f(t,B,C,D) Let's look at each of the terms. E and A↲5 are easy. Wt is the t th 32-bit word in the 80-word block. Kt is a constant, but it varies according to which word you're on (do you like the concept of a variable constant?): Kt = □□ = 5a82799916 (0 <= t <= 19) Kt = □□ = 6ed9eba116 (20 <= t <= 39) Kt = □□ = 8f1bbcdc16 (40 <= t <= 59) Kt = □□ = ca62c1d616 (60 <= t <= 79) ⊕ 16 words of message 160-bit intermediate MD value ABCDE generated data complicated function ABCDE Figure 5-9. Inner Loop of SHA-1—80 Iterations per Block ↲ 30 ↲ 1 in revised version 2 2 30 2 3 30 2 5 30 2 10 30 136 HASHES AND MESSAGE DIGESTS 5.7 f(t,B,C,D) is a function that varies according to which of the eighty words you're working on: 5.7 HMAC The idea of using a message digest algorithm in the construction of a MAC algorithm was described in section §5.2.2 Computing a MAC with a Hash. Given that one intuitively reasonable approach—computing the digest of the concatenation of a shared secret with the message—has flaws, cryptographers set out to find a construction that could be proven secure. Proving security of a cryptographic algorithm is difficult. First you have to start by defining "secure". Then the best you can do is prove that one algorithm is at least as secure as another. HMAC resulted from an effort to find a MAC algorithm that could be proven to be secure if the underlying message digest's compression function (see §5.4.2 Overview of MD4 Message Digest Computation) was secure. They defined secure as having two properties: • collision resistance (infeasible to find two inputs that yield the same output) • an attacker that doesn't know the key K cannot compute the proper digest(K,x) for data x, even if the attacker can see the value of digest(K,y), for arbitrary numbers of inputs y, with y not equal to x. So in the case of HMAC, they proved that HMAC was secure (had those 2 properties) provided that the underlying compression function was secure (had those 2 properties). Although there are likely other alternatives that are more efficient and just as secure, nobody has offered the same sort of proof of those alternatives. So HMAC seems likely to become the de facto standard. In essence, HMAC prepends the key to the data, digests it, and then prepends the key to the result and digests that. This nested digest with secret inputs to both iterations prevents the extension attacks that would be possible if you simply digested the key and message once. In detail, HMAC function takes a variable-length key and a variable-sized message and produces a fixed-size output that is the same size as the output of the underlying digest. It first pads the key with 0 bits to 512 bits. If the key is larger than 512 bits, then HMAC first digests the key, resulting in 128 bits or 160 bits (depending on the size of output of the digest function) and pads the result out to 512 bits. It f(t,B,C,D) = (B ∧ C) ∨ (∼B ∧ D) (0 <= t <= 19) f (t,B,C,D) = B ⊕ C ⊕ D (20 <= t <= 39) f(t,B,C,D) = (B ∧ C) ∨ (B ∧ D) ∨ (C ∧ D) (40 <= t <= 59) f (t,B,C,D) = B ⊕ C ⊕ D (60 <= t <= 79) 5.8 HOMEWORK 137 then ⊕s the padded key with a constant string of octets of value 3616, concatenates it with the message to be protected and computes a message digest. It ⊕s the padded key with a different constant string of octets of value 5c16, concatenates that with the result of the first digest, and computes a second digest on the result. 5.8 HOMEWORK 1. Doing a signature with RSA alone on a long message would be too slow (presumably using cipher block chaining). Suppose we could do division quickly. Would it be reasonable to compute an RSA signature on a long message by first finding what the message equals, mod n, and signing that? 2. Message digests are reasonably fast, but here's a much faster function to compute. Take your message, divide it into 128-bit chunks, and ⊕ all the chunks together to get a 128-bit result. Do the standard message digest on the result. Is this a good message digest function? key 0 const2 ⊕ ⊕ const1 message digest digest HMAC(key, message) Figure 5-10. HMAC 138 HASHES AND MESSAGE DIGESTS 5.8 3. In §5.1 Introduction we discuss the devious secretary Bob having an automatic means of generating many messages that Alice would sign, and many messages that Bob would like to send. By the birthday problem, by the time Bob has tried a total

of 232 messages, he will probably have found two with the same message digest. The problem is, both may be of the same type, which would not do him any good. How many messages must Bob try before it is probable that he'll have messages with matching digests, and that the messages will be of opposite types? 4. In §5.2.4.2 Hashing Large Messages, we described a hash algorithm in which a constant was successively encrypted with blocks of the message. We showed that you could find two messages with the same hash value in about 232 operations. So we suggested doubling the hash size by using the message twice, first in forward order to make up the first half of the hash, and then in reverse order for the second half of the hash. Assuming a 64-bit encryption block, how could you find two messages with the same hash value in about 232 iterations? Hint: consider blockwise palindromic messages. 5. Design a modification to MD2 to handle messages which are not an integral number of octets. Design it so that messages that are an integral number of octets have the same digest value as with the existing MD2. 6. Why do MD4, MD5, and SHA-1 require padding of messages that are already a multiple of 512 bits? 7. Modify the specification of SHA-1 so that it looks a lot more like MD4 and MD5. Do this by having each of the words A, B, C, D, and E modified in place rather than (as SHA-1 specifies it) modifying A and then basically rotating the words. Alternatively, modify the specifications of MD4 and MD5 to make them look more like SHA-1. How would you choose which specification to base an implementation on in a particular underlying architecture? 8. Open-ended project: Implement one or more of the message digest algorithms and test how "random" the output appears. For example, test the percentage of 1 bits in the output, or test how many bits of output change with minor changes in the input. Also, design various simplifications of the message digest functions (such as reducing the number of rounds) and see how these change things. 9. What are the minimal and maximal amounts of padding that would be required in each of the message digest functions? 10. Assume ∧, ∨, ⊕, +, ∼, and ↲ all take about the same amount of time. Estimate the relative performance of MD2, MD4, MD5, and SHA-1. 11. Show that the checksum function in MD2 would not be a good message digest function by showing how to generate a message with a given checksum. 5.8 HOMEWORK 139 12. Assume a good 128-bit message digest function. Assume there is a particular value, d, for the message digest and you'd like to find a message that has a message digest of d. Given that there are many more 2000-bit messages that map to a particular 128-bit message digest than 1000-bit messages, would you theoretically have to test fewer 2000-bit messages to find one that has a message digest of d than if you were to test 1000-bit messages? 13. Why do we expect that a randomly chosen 100-bit number will have about the same number of 1 bits and 0 bits? (For you statistics fans, calculate the mean and standard deviation of the number of 1 bits.) 14. For purposes of this exercise, we will define random as having all elements equally likely to be chosen. So a function that selects a 100-bit number will be random if every 100-bit number is equally likely to be chosen. Using this definition, if we look at the function "+" and we have two inputs, x and y, then the output will be random if at least one of x and y are random. For instance, y can always be 51, and yet the output will be random if x is random. For the following functions, find sufficient conditions for x, y, and z under which the output will be random: ∼x x⊕y x∨y x∧y (x∧y)∨(∼x∧z) [the selection function] (x∧y)∨(x∧z)∨(y∧z) [the majority function] x⊕y⊕z y⊕(x∨∼z) 15. Prove that the function (x∧y)⊕(x∧z)⊕(y∧z) and the function (x∧y)∨(x∧z)∨(y∧z) are equivalent. (Sorry—this isn't too relevant to cryptography, but we'd stumbled on two different versions of this function in different documentation and we had to think about it for a bit to realize they were the same. We figured you should have the same fun.) 16. We mentioned in §5.2.2 Computing a MAC with a Hash that using MD4(KAB|m) as a MAC is not secure. This is not a problem if MD2 is used instead of MD4. Why is that the case? 17. In §5.2.3.1 Generating a One-Time Pad, we generate a pseudo-random stream of MD-sized

blocks. This stream must eventually repeat (since only 2MD-size different blocks can be generated). Will the first block necessarily be the first to be repeated? How does this compare to OFB (see Chapter 4 Modes of Operation Homework Problem 2)? 18. How do you decrypt the encryption specified in §5.2.3.2 Mixing In the Plaintext? 140 HASHES AND MESSAGE DIGESTS 5.8 19. Can you modify the encryption specified in §5.2.3.2 Mixing In the Plaintext so that instead of bi = MD(KAB|ci−1) we use bi = MD(KAB|pi−1)? How do you decrypt it? Why wouldn't the modified scheme be as secure? (Hint: what would happen if the plaintext consisted of all zeroes?) 141 6 PUBLIC KEY ALGORITHMS 6.1 INTRODUCTION This chapter describes public key cryptography. To really understand how and why the public key algorithms work, it is necessary to know some number theory. In this chapter we'll describe the number theory concepts necessary to understand the public key algorithms, but only in sufficient detail for an intuitive understanding. For instance, we won't give proofs, though we will explain things. And in the case of complex algorithms (like Euclid's algorithm), we'll merely state that the algorithm exists and what it does, rather than describing the algorithm in this chapter. Those desiring a more complete treatment of the subject matter can read Chapter 23 Number Theory. Public key algorithms are a motley crew. All the hash algorithms do the same thing—they take a message and perform an irreversible transformation on it. All the secret key algorithms do the same thing—they take a block and encrypt it in a reversible way, and there are chaining methods to convert the block ciphers into message ciphers. But public key algorithms look very different from each other, not only in how they perform their functions, but in what functions they perform. We'll describe: • RSA and ECC, which do encryption and digital signatures • ElGamal and DSS, which do digital signatures • Diffie-Hellman, which allows establishment of a shared secret but doesn't have any algorithms that actually use the secret (it would be used, for instance, together with a secret key scheme in order to actually use the secret for something like encryption) • zero knowledge proof systems, which only do authentication The thing that all public key algorithms have in common is the concept of a pair of related quantities, one secret and one public, associated with each principal. (A principal is anything or anyone participating in cryptographically protected communication.) 142 PUBLIC KEY ALGORITHMS 6.2 6.2 MODULAR ARITHMETIC There was a young fellow named Ben Who could only count modulo ten. He said, "When I go Past my last little toe I shall have to start over again." —Anonymous Most of the public key algorithms are based on modular arithmetic. Modular arithmetic uses the non-negative integers less than some positive integer n, performs ordinary arithmetic operations such as addition and multiplication, and then replaces the result with its remainder when divided by n. The result is said to be modulo n or mod n. When we write "x mod n", we mean the remainder of x when divided by n. Sometimes we'll leave out "mod n" when it's clear from context. 6.2.1 Modular Addition Let's look at mod 10 addition. 3 + 5 = 8, just like in regular arithmetic. The answer is already between 0 and 9. 7 + 6 = 13 in regular arithmetic, but the mod 10 answer is 3. Basically, one can perform mod 10 arithmetic by using the last digit of the answer. For example, 5 + 5 = 0 3 + 9 = 2 2 + 2 = 4 9 + 9 = 8 Let's look at the mod 10 addition table (Figure 6-1). Addition of a constant mod 10 can be used as a scheme for encrypting digits, in that it maps each decimal digit to a different decimal digit in a way that is reversible; the constant is our secret key. It's not a good cipher, of course, but it is a cipher. (It's actually a Caesar cipher.) Decryption would be done by subtracting the secret key modulo 10, which is an easy operation—just do ordinary subtraction and if the result is less than 0, add 10. Just like in regular arithmetic, subtracting x can be done by adding −x, also known as x's additive inverse. An additive inverse of x is the number you'd have to add to x to get 0. For example, 4's inverse will be 6, because in mod 10 arithmetic 4 + 6 = 0. If the secret key were 4, then to encrypt we'd add 4 (mod 10), and to decrypt we'd add 6 (mod 10). 6.2.2 MODULAR ARITHMETIC 143 6.2.2

Modular Multiplication Now let's look at the mod 10 multiplication table (Figure 6-2). Multiplication by 1, 3, 7, or 9 works as a cipher, because it performs a one-to-one substitution of the digits. But multiplication by any of the other numbers will not work as a cipher. For instance, if you tried to encrypt by multiplying by + 0123456789 00123456789 11234567890 22345678901 33456789012 44567890123 55678901234 66789012345 77890123456 88901234567 99012345678 Figure 6-1. Addition Modulo 10 · 0123456789 00000000000 10123456789 20246802468 30369258147 40482604826 50505050505 60628406284 70741852963 80864208642 90987654321 Figure 6-2. Multiplication Modulo 10 144 PUBLIC KEY ALGORITHMS 6.2.2 5, half the numbers would encrypt to 0 and the other half would encrypt to 5. You've lost information. You can't decrypt the ciphertext 5, since the plaintext could be any of {1, 3, 5, 7, 9}. So multiplication mod 10 can be used for encryption, provided that you choose the multiplier wisely. But how do you decrypt? Well, just like with addition, where we undid the addition by adding the additive inverse, we'll undo the multiplication by multiplying by the multiplicative inverse. In ordinary arithmetic, x's multiplicative inverse is 1/x. If x is an integer, then its multiplicative inverse is a fraction. In modular arithmetic though, the only numbers that exist are integers. The multiplicative inverse of x (written x^{-1}) is the number by which you'd multiply x to get 1. Only the numbers {1, 3, 7, 9} have multiplicative inverses mod 10. For example, 7 is the multiplicative inverse of 3. So encryption could be performed by multiplying by 3, and decryption could be performed by multiplying by 7. 9 is its own inverse. And 1 is its own inverse. Multiplication mod n is not a secure cipher, but it works, in the sense that we can scramble the digits by multiplying by x and get back to the original digits by multiplying by x^{-1}. It is by no means obvious how you find a multiplicative inverse in mod n arithmetic, especially if n is very large. For instance if n was a 100-digit number, you would not be able to do a brute-force search for an inverse. But it turns out there is an algorithm that will efficiently find inverses mod n. It is known as Euclid's algorithm. §23.4 Euclid's Algorithm gives the details of the algorithm. For here, all you need to know is what it does. Given x and n, it finds the number y such that $x \cdot y \bmod n = 1$ (if there is one). What's special about the numbers {1, 3, 7, 9}? Why is it they're the only ones, mod 10, with multiplicative inverses? The answer is that those numbers are all relatively prime to 10. Relatively prime means they do not share any common factors other than 1. For instance, the largest integer that divides both 9 and 10 is 1. The largest integer that divides both 7 and 10 is 1. In contrast, 6 is not one of {1, 3, 7, 9}, and it does not have a multiplicative inverse mod 10. It's also not relatively prime to 10 because 2 divides both 10 and 6. In general, when we're working mod n, all the numbers relatively prime to n will have multiplicative inverses, and none of the other numbers will. And mod n multiplication by any number x relatively prime to n will work as a cipher because we can multiply by x to encrypt, and then multiply by x^{-1} to decrypt. (Again let us hasten to reassure you that we're not claiming it's a good cipher, in the sense of being secure. What we mean by its being a cipher is that we can modify the information through one algorithm (multiplication by x mod n) and then reverse the process (by multiplying by x^{-1} mod n). How many numbers less than n are relatively prime to n? Why would anyone care? Well, it turns out to be so useful that it's been given its own notation—$\phi(n)$. ϕ is called the totient function, supposedly from total and quotient. How big is $\phi(n)$? If n is prime, then all the integers {1, 2,...n−1} are relatively prime to n, so $\phi(n) = n-1$. If n is a product of two distinct primes, say p and q, then there are $(p-1)(q-1)$ numbers relatively prime to n, so $\phi(n) = (p-1)(q-1)$. Why is that? Well, there are $n = pq$ total numbers in {0, 1, 2,...n−1}, and we want to exclude those numbers that aren't relatively prime to n. Those are the numbers that are either multiples of p or of q. There are p multiples of q less than pq and q multiples of p less than pq. So there are $p+q-1$ numbers less than 6.2.3 MODULAR ARITHMETIC 145 pq that aren't relatively prime to pq (we can't count 0 twice!). Thus $\phi(pq) = pq - (p+q-1) = (p-1)(q-1)$. 6.2.3

Modular Exponentiation Modular exponentiation is again just like ordinary exponentiation. Once you get the answer, you divide by n and get the remainder. For instance, 46 = 6 mod 10 because 46 = 4096 in ordinary arithmetic, and 4096 = 6 mod 10. Let's look at the exponentiation table mod 10. We are purposely putting in extra columns because in exponentiation, xy mod n is not the same as xy+n mod n. For instance, 31 = 3 mod 10, but 311 = 7 mod 10 (it's 177147 in ordinary arithmetic). Let's look at mod 10 exponentiation (Figure 6-3). Note that exponentiation by 3 would act as an encryption of the digits, in that it rearranges all the digits. Exponentiation by 2 would not, because both 22 and 82 are 4 mod 10. How would you decrypt? Is there an exponentiative inverse like there is a multiplicative inverse? Just like with multiplication, the answer is sometimes. (I bet people are going to hate our inventing the word exponentiative, but it's a useful word.) Now we'll throw in an amazing fact about ϕ(n). Looking at the exponentiation table, we notice that columns 1 and 5 are the same, and 2 and 6 are the same, and 3 and 7 are the same. It turns out that xy mod n is the same as x(y mod ϕ(n)) mod n. In the case of 10, the numbers relatively prime to 10 are {1, 3, 7, 9}, so ϕ(n) = 4. So that's why the i th column is the same as the i+4th column. (Note for picky mathematicians—this fact isn't true for all n, but it's true for all n we care xy 0 1 2 3 4 5 6 7 8 9 10 11 12 0 000000000000 11111111111111 21248624862486 31397139713971 41464646464646 51555555555555 61666666666666 71793179317931 81842684268426 91919191919191 Figure 6-3. Exponentiation Modulo 10 146 PUBLIC KEY ALGORITHMS 6.3 about. It's true for primes and it's true for any product of distinct primes, i.e. it's true for any n that doesn't have p2 as a factor for any prime p—such an n is known as square free.) What we'll find important is the special case of this where y = 1 mod ϕ(n), i.e. if y = 1 mod ϕ(n), then for any number x, x y = x mod n. Armed with this knowledge, let's look at RSA. 6.3 RSA RSA is named after its inventors, Rivest, Shamir, and Adleman. It is a public key cryptographic algorithm that does encryption as well as decryption. The key length is variable. Anyone using RSA can choose a long key for enhanced security, or a short key for efficiency. The most commonly used key length for RSA is 512 bits. The block size in RSA (the chunk of data to be encrypted) is also variable. The plaintext block must be smaller than the key length. The ciphertext block will be the length of the key. RSA is much slower to compute than popular secret key algorithms like DES and IDEA. As a result, RSA does not tend to get used for encrypting long messages. Mostly it is used to encrypt a secret key, and then secret key cryptography is used to actually encrypt the message. 6.3.1 RSA Algorithm First, you need to generate a public key and a corresponding private key. Choose two large primes p and q (probably around 256 bits each). Multiply them together, and call the result n. The factors p and q will remain secret. (You won't tell anybody, and it's practically impossible to factor numbers that large.) To generate your public key, choose a number e that is relatively prime to ϕ(n). Since you know p and q, you know ϕ(n)—it's (p−1)(q−1). Your public key is ⟨e,n⟩. To generate your private key, find the number d that is the multiplicative inverse of e mod ϕ(n). ⟨d,n⟩ is your private key. To encrypt a message m (< n), someone using your public key should compute ciphertext c = me mod n. Only you will be able to decrypt c, using your private key to compute m = cd mod n. Also, only you can sign a message m (< n) with signature s = md mod n based on your private key. Anyone can verify your signature by checking that m = se mod n. That's all there is to RSA. Now there are some questions we should ask. • Why does it work? (E.g., will decrypting an encrypted message get the original message back?) 6.3.2 RSA 147 • Why is it secure? (E.g., given e and n, why can't someone easily compute d?) • Are the operations encryption, decryption, signing, and verifying signatures all sufficiently efficient to be practical? • How do we find big primes? 6.3.2 Why Does RSA Work? RSA does arithmetic mod n, where n = pq. We know that ϕ(n) = (p−1)(q−1). We've chosen d and e such that de = 1 mod ϕ(n). Therefore, for any x, xde = x mod n. An RSA encryption consists of taking x and raising it to e. If we take the

result and raise it to the d (i.e., perform RSA decryption), we'll get (xe) d, which equals xed, which is the same as x. So we see that decryption reverses encryption. In the case of signature generation, x is first raised to the d power to get the signature and then the signature is raised to the e power for verification; the result, xde, will equal x. 6.3.3 Why Is RSA Secure? We don't know for sure that RSA is secure. We can only depend on the Fundamental Tenet of Cryptography—lots of smart people have been trying to figure out how to break RSA, and they haven't come up with anything yet. The real premise behind RSA's security is the assumption that factoring a big number is hard. The best known factoring methods are really slow. To factor a 512-bit number with the best known techniques would take about thirty thousand MIPS-years [ROBS95]. We suspect that a better technique is to wait a few years and then use the best known technique. If you can factor quickly, you can break RSA. Suppose you are given Alice's public key ⟨e,n⟩. If you could find e's exponentiative inverse mod n, then you'd have figured out Alice's private key ⟨d,n⟩. How can you find e's exponentiative inverse? Alice did it by knowing the factors of n, allowing her to compute ϕ(n). She found the number that was e's multiplicative inverse mod ϕ(n). She didn't have to factor n—she started with primes p and q and multiplied them together to get n. You can do what Alice did if you can factor n to get p and q. We do not know that factoring n is the only way of breaking RSA. We know that breaking RSA (for example, having an efficient means of finding d, given e and n) is no more difficult than factoring [CORM91], but there might be some other means of breaking RSA. Note that it's possible to misuse RSA. For instance, let's say I'm going to send Alice a message divulging the name of the Cabinet member who allegedly once hired a kid to mow his/her lawn, and didn't fill out all the proper IRS forms. Bob knows that's what I'm going to transmit. I'll 148 PUBLIC KEY ALGORITHMS 6.3.4 encrypt the text string which is the guilty person's name using Alice's public key. Bob can't possibly decrypt it, because we believe RSA is secure. So what can Bob learn from eavesdropping on the encrypted data? Well, Bob can't decrypt, but he can encrypt. He knows I'm sending one of fourteen possible messages. He takes each Cabinet member's name and encrypts it with Alice's public key. One of them will match my message—unless I use RSA properly. In §6.3.6 Public-Key Cryptography Standard (PKCS) we'll discuss how to use RSA properly. For now, a simple thing I can do to prevent Bob from guessing my message, encrypting with Alice's public key, and checking the result, is to concatenate the name with a large random number, say 64 bits long. Then instead of fourteen possible messages for Bob to check, there are 14×264, and checking that many messages is computationally infeasible. 6.3.4 How Efficient Are the RSA Operations? The operations that need to be routinely performed with RSA are encryption, decryption, generating a signature, and verifying a signature. These need to be very efficient, because they will be used a lot. Finding an RSA key (which means picking appropriate n, d, and e) also needs to be reasonably efficient, but it isn't as critical as the other operations, since it is done less frequently. As it turns out, finding an RSA key is substantially more computationally intensive than using one. 6.3.4.1 Exponentiating with Big Numbers Encryption, decryption, signing, and verifying signatures all involve taking a large number, raising it to a large power, and finding the remainder mod a large number. For the sizes the numbers have to be for RSA to be secure, these operations would be prohibitively expensive if done in the most straightforward way. The following will illustrate some tricks for doing the calculation faster. Suppose you want to compute 12354 mod 678. The straightforward thing to do (assuming your computer has a multiple-precision arithmetic package) is to multiply 123 by itself 54 times, getting a really big product (about 100 digits), and then to divide by 678 to get the remainder. A computer could do this with ease, but for RSA to be secure, the numbers must be on the order of 150 digits. Raising a 150-digit number to a 150-digit power by this method would exhaust the capacity of all existing computers for more than the expected life of the universe,

and thus would not be cost-effective. Luckily, you can do better than that. If you do the modular reduction after each multiply, it keeps the number from getting really ridiculous. To illustrate: 1232 = 123 · 123 = 15129 = 213 mod 678 6.3.4.1 RSA 149 This reduces the problem to 54 small multiplies and 54 small divides, but it would still be unacceptable for exponents of the size used with RSA. However, there is a much more efficient method. To raise a number x to an exponent which is a power of 2, say 32, you could multiply by x 32 times, which is reasonable if you have nothing better to do with your time. A much better scheme is to first square x, then square the result, and so on. Then you'll be done after 5 squarings (5 multiplies and 5 divides): What if you're not lucky enough to be raising something to a power of 2? First note that if you know what 123x is, then it's easy to compute 1232x —you get that by squaring 123x . It's also easy to compute 1232x+1—you get that by multiplying 1232x by 123. Now you use this observation to compute 12354. Well, 54 is 1101102 (represented in binary). You'll compute 123 raised to a sequence of powers—12, 112, 1102, 11012, 110112, 1101102. Each successive power concatenates one more bit of the desired exponent. And each successive power is either twice the preceding power or one more than twice the preceding power: 1233 = 123 · 213 = 26199 = 435 mod 678 1234 = 123 · 435 = 53505 = 621 mod 678 1232 = 123 · 123 = 15129 = 213 mod 678 1234 = 213 · 213 = 45369 = 621 mod 678 1238 = 621 · 621 = 385641 = 537 mod 678 12316 = 537 · 537 = 288369 = 219 mod 678 12332 = 219 · 219 = 47961 = 501 mod 678 1232 = 123·123 = 15129 = 213 mod 678 1233 = 1232·123 = 213·123 = 26199 = 435 mod 678 1236 = (1233) 2 = 4352 = 189225 = 63 mod 678 12312 = (1236) 2 = 632 = 3969 = 579 mod 678 12313 = 12312·123 = 579·123 = 71217 = 27 mod 678 12326 = (12313) 2 = 27 2= 729 = 51 mod 678 12327 = 12326·123 = 51·123 = 6273 = 171 mod 678 12354 = (12327) 2 = 1712 = 29241 = 87 mod 678 150 PUBLIC KEY ALGORITHMS 6.3.4.2 In other words, raising 123 to the 54 can be done by repeated squaring, together with sporadic multiplication by 123 for the bits that are 1: The idea is that squaring is the same as multiplying the exponent by two, which in turn is the same as shifting the exponent left by one bit. And multiplying by the base is the same as adding one to the exponent. In general, to perform exponentiation of a base to an exponent, you start with your value set to 1. As you read the exponent in binary bit by bit from high-order bit to low-order bit, you square your value, and if the bit is a 1 you then multiply by the base. You perform modular reduction after each operation to keep the intermediate results small. By this method you've reduced the computation of 12354 to 8 multiplies and 8 divides. More importantly, the number of multiplies and divides rises linearly with the length of the exponent in bits rather than with the value of the exponent itself. RSA operations using this technique are sufficiently efficient to be practical. 6.3.4.2 Generating RSA Keys Most uses of public key cryptography do not require frequent generation of RSA keys. If generation of an RSA key is only done, for instance, when an employee is hired, then it need not be as efficient as the operations that use the keys. However, it still has to be reasonably efficient. 6.3.4.2.1. Finding Big Primes p and q There is an infinite supply of primes. However, they thin out as numbers get bigger and bigger. The probability of a randomly chosen number n being prime is approximately 1/ln n. The natural logarithm function, ln, rises linearly with the size of the number represented in digits or bits. For a ten-digit number, there is about one chance in 23 of it being prime. For a hundred-digit number (a size that would be useful for RSA), there is about one chance in 230. So, we'll choose a random number, and test if it is prime. On the average, we'll only have to try 230 of them before we find one that is a prime. So, how do we test if a number n is prime? One naive method is to divide n by all numbers ≤ and see if the division comes out even. The problem is, that would take several universe lifetimes for each candidate prime. We said finding p and q didn't need to be as easy as generating or verifying a signature, but forever is too long. 54 = (((((1)2+1)2)2+1)2+1)2, so 12354 = (((((123)2 123)2) 2123)2 123)2 = 87 mod 678. n 6.3.4.2 RSA

151 It turns out there is no known practical way for absolutely determining that a number of this size is prime. Fortunately, there is a test for determining that a number is probably prime, and the more time we spend testing a number the more assured we can be that the number is prime. We'll use Euler's Theorem: For any a relatively prime to n, $a\phi(n) = 1 \bmod n$. (See §23.8 Euler's Theorem for a proof.) In the case where n is a prime, $\phi(n) = n - 1$. The theorem then takes on a simpler form and in fact another name: Fermat's Theorem: If p is prime and $0 < a < p$, $a^{p-1} = 1 \bmod p$. You might ask the question (somebody did)—does $a^{n-1} = 1 \bmod n$ hold even when n is not prime? The answer is—usually not! A primality test, then, for a number n is to pick a number $a < n$, compute $a^{n-1} \bmod n$, and see if the answer is 1. If it is not 1, n is certainly not prime. If it is 1, n may or may not be prime. If n is a randomly generated number of about a hundred digits, the probability that n isn't prime but $a^{n-1} \bmod n = 1$ is about 1 in 1013 [POME81, CORM91]. Most people would decide they could live with that risk of falsely assuming n was prime when it wasn't. The cost of such a mistake would be that (1) RSA might fail—they could not decrypt a message addressed to them, or (2) someone might be able to compute their private exponent with less effort than anticipated. There aren't many applications where a risk of failure of 1 in 1013 is a problem. But if the risk of 1 in 1013 is unacceptable, the primality test can be made more reliable. A likely thing to try is using multiple values of a. If for any given n, each value of a had a probability of 1 in 1013 of falsely reporting primality, a few tests would assure even the most paranoid person. Unfortunately, there exist numbers n which are not prime, but which satisfy $a^{n-1} = 1 \bmod n$ for all values of a. They are called Carmichael numbers. Carmichael numbers are sufficiently rare that the chance of selecting one at random is nothing to lose sleep over. Nevertheless, mathematicians have come up with an enhancement to the above primality test that will detect non-primes (even Carmichael numbers) with high probability and negligible additional computation, so we may as well use it. The method of choice for testing whether a number is prime is due to Miller and Rabin [RABI80]. We can always express $n-1$ as a power of two times an odd number, say $2^b c$. We can then compute $a^{n-1} \bmod n$ by computing $a^c \bmod n$ and then squaring the result b times. If the result is not 1, then n is not prime and we're done. If the result is 1, we can go back and look at those last few intermediate squarings. (If we're really clever, we'll be checking the intermediate results as we compute them.) If $a^c \bmod n$ is not 1, then one of the squarings took a number that was not 1 and squared it to produce 1. That number is a mod n square root of 1. It turns out that if n is prime, then the only mod n square roots of 1 are 1 and −1 (also known as n−1). Further, if n is not a power of a prime, then 1 has many square roots, and all are equally likely to be found by this test. For more on why, see §23.5 Chinese Remainder Theorem. So if the Miller-Rabin test finds a square root of 1 that is not ±1, then n is not prime. Furthermore, if n is not prime (even if it is a Carmichael number), at least ¾ of all possible values of a will show n to be composite. By trying many values for a, we can 152 PUBLIC KEY ALGORITHMS 6.3.4.3 make the probability of falsely identifying n as prime inconceivably small. In actual implementations, how many values of a to try is a trade-off between performance and paranoia. To summarize, an efficient method of finding primes is: 1. Pick an odd random number n in the proper range. 2. Test n's divisibility by small primes and go back to step 1 if you find a factor. (Obviously, this step isn't necessary, but it's worth it since it has a high enough probability of catching some non-primes and is much faster than the next step). 3. Repeat the following until n is proven not prime (in which case go back to step 1) or as many times as you feel necessary to show that n is probably prime: Pick an a at random and compute $a^c \bmod n$ (where c is the odd number for which $n-1 = 2^b c$). During the computation of $a^c \bmod n$, each time mod n squaring is performed, check if the result is 1; if so, check if the number that was squared (which is a square root of 1) is ±1; if not, n is not prime. Next, if the result of the computation of $a^c \bmod n$ is ±1, n passes the primality test for this a.

Otherwise, at most b−1 times, replace the result by its square and check if it is ±1. If it is 1, n is not prime (because the previous result is a square root of 1 different from ±1). If it is −1, n passes the primality test for this a. If you've done the squaring b−1 times, n is not prime (because a(n−1)/2 is not ±1). 6.3.4.2.2. Finding d and e How do we find d and e given p and q? As we said earlier, for e we can choose any number that is relatively prime to (p−1)(q−1), and then all we need to do is find the number d such that ed = 1 mod ϕ(n). This we can do with Euclid's algorithm. There are two strategies one can use to ensure that e and (p−1)(q−1) are relatively prime. 1. After p and q are selected, choose e at random. Test to see if e is relatively prime to (p−1)(q−1). If not, select another e. 2. Don't pick p and q first. Instead, first choose e, then select p and q carefully so that (p−1) and (q−1) are guaranteed to be relatively prime to e. The next section will explain why you'd want to do this. 6.3.4.3 Having a Small Constant e A rather astonishing discovery is that RSA is no less secure (as far as anyone knows) if e is always chosen to be the same number. And if e is chosen to be small, or easy to compute, then the operations of encryption and signature verification become much more efficient. Given that the proce- 6.3.4.3 RSA 153 dure for finding a d and e pair is to pick one and then derive the other, it is straightforward to make e be a small constant. This makes public key operations faster while leaving private key operations unchanged. You might wonder whether it would be possible to select small values for d to make private key operations fast at the expense of public key operations. The answer is that you can't. If d were a constant, the scheme would not be secure because d is the secret. If d were small, an attacker could search small values to find d. Two popular values of e are 3 and 65537. Why 3? 2 doesn't work because it is not relatively prime to (p−1)(q−1) (which must be even because p and q are both odd). 3 can work, and with 3, public key operations require only two multiplies. Using 3 as the public exponent maximizes performance. As far as anyone knows, using 3 as a public exponent does not weaken the security of RSA if some practical constraints on its use are followed. Most dramatically, if a message m to be encrypted is small—in particular, smaller than —then raising m to the power of three and reducing mod n will simply produce the value m3 . Anyone seeing such an encrypted message could decrypt it simply by taking a cube root. This problem can be avoided by padding each message with a random number before encryption, so that m3 is always large enough to be guaranteed to need to be reduced mod n. A second problem with using 3 as an exponent is that if the same message is sent encrypted to three or more recipients each of whom has a public exponent of 3, the message can be derived from the three encrypted values and the three public keys ⟨3, n1⟩, ⟨3, n2⟩, ⟨3, n3⟩. Suppose a bad guy sees m3 mod n1, m3 mod n2, and m3 mod n3 and knows ⟨3, n1⟩, ⟨3, n2⟩, ⟨3, n3⟩. Then by the Chinese Remainder computation (see §23.5 Chinese Remainder Theorem), the bad guy can compute m3 mod n1n2n3. Since m is smaller than each of the nis (because RSA can only encrypt messages smaller than the modulus), m3 will be smaller than n1n2n3, so m3 mod n1n2n3 will just be m3. Therefore, the bad guy can compute the ordinary cube root of m3 (which again is easy if you are a computer), giving m. Now this isn't anything to get terribly upset about. In practical uses of RSA, the message to be encrypted is usually a key for a secret key encryption algorithm and in any case is much smaller than n. As a result, the message must be padded before it is encrypted. If the padding is randomly chosen (and it should be for a number of reasons), and if it is rechosen for each recipient, then there is no threat from an exponent of 3 no matter how many recipients there are. The padding doesn't really have to be random—for example, the recipient's ID would work fine. Finally, an exponent of 3 works only if 3 is relatively prime to ϕ(n) (in order for it to have an inverse d). How do we choose p and q so that 3 will be relatively prime to ϕ(n) = (p−1)(q−1)? Clearly, (p−1) and (q−1) must each be relatively prime to 3. To ensure that p−1 is relatively prime to 3, we want p to be 2 mod 3. That will ensure p−1 is 1 mod 3. Similarly

we want q to be 2 mod 3. We can make sure that the only primes we select are congruent to 2 mod 3 by choosing a random number, multiplying by 3 and adding 2, and using that as the number we will test for primality. Indeed, we want to make sure the number we test is odd (since if it's even it is unlikely to be n 3 154 PUBLIC KEY ALGORITHMS 6.3.4.4 prime), so we should start with an odd number, multiply by 3 and add 2. This is equivalent to starting with any random number, multiplying by 6 and then adding 5. Another popular value of e is 65537. Why 65537? The appeal of 65537 (as opposed to others of the same approximate size) is that 65537 = 216+1 and it is prime. Because its binary representation contains only two 1s, it takes only 17 multiplies to exponentiate. While this is much slower than the two multiplies required with an exponent of 3, it is much faster than the 768 (on average) required with a randomly chosen 512-bit value (the typical size of an RSA modulus in practical use today). Also, using the number 65537 as a public exponent largely avoids the problems with the exponent 3. The first problem with 3 occurs if m3 < n. Unless n is much longer than the 512 bits in typical use today, there aren't too many values of m for which m65537 < n, so being able to take a normal 65537th root is not a threat. The second problem with 3 occurs when the same message is sent to at least 3 recipients. In theory, with 65537 there is a threat if the same message is sent encrypted to at least 65537 recipients. A cynic would argue that under such circumstances, the message couldn't be very secret. The third problem with 3 is that we have to choose n so that ϕ(n) is relatively prime to 3. For 65537, the easiest thing to do is just reject any p or q which is equal to 1 mod 65537. The probability of rejection is very small (2−16), so this doesn't make finding n significantly harder. 6.3.4.4 Optimizing RSA Private Key Operations There is a way to speed up RSA exponentiations in generating signatures and decrypting (the operations using the private key) by taking advantage of knowledge of p and q. Feel free to skip this section—it isn't a prerequisite for anything else in the book. And it requires more than the usual level of concentration. In RSA, d and n are on the order of 512-bit numbers, or 150 digits. p and q are on the order of 256 bits, or 75 digits. RSA private key operations involve taking some c (usually a 512-bit number) and computing cd mod n. It's easy to say "raise a 512-bit number to a 512-bit exponent mod a 512-bit number," but it's certainly processor-intensive, even if you happen to be a silicon-based computer. A way to speed up RSA operations is to do all the computation mod p and mod q, then use the Chinese Remainder Theorem to compute what the answer is mod pq. So suppose you want to compute m = cd mod n. Instead, you could take cp = c mod p and cq = c mod q and compute mp = cp d mod p and mq = cq d mod q, then use the Chinese Remainder Theorem to convert back to what m would equal mod n, which would give you cd mod n. Also, it is not necessary to raise to the dth power mod p, given that d is going to be bigger than p (by a factor of about q). Since (by Euler's Theorem) any ap−1 = 1 mod p, we can take d's value mod p−1 and use that as the exponent instead. In other words, if d = k(p−1) + r, then cd mod p = cr mod p. 6.3.5 RSA 155 So, let us compute dp = d mod (p−1) and dq = d mod (q−1). Then, instead of doing the expected RSA operation of m = cd mod n which involves 512-bit numbers, we'll compute both mp = cp dp mod p and mq = cq dq mod q and then compute m from the Chinese Remainder Theorem. To save ourselves work, since we'll be using dp and dq a lot (every time we do an RSA private key computation), we'll compute them once and remember them. Similarly, to use the Chinese Remainder Theorem at the end, we need to know p−1 mod q and q−1 mod p, so we'll precompute and remember them as well. All told, instead of one 512-bit exponentiation, this modified calculation does two 256-bit exponentiations, followed by two 256-bit multiplies and a 512-bit add. This might not seem like a net gain, but because the exponents are half as long, using this variant makes RSA about twice as fast. Note that to do these optimizations for RSA operations, we need to know p and q. Someone who is only supposed to know the public key will not know p and q (or else they can easily compute d).

Therefore, these optimizations are only useful for the private key operations (decryption and generating signatures). However, that's okay because we can choose e to be a convenient value (like 3 or 65537) so that raising a 512-bit number to e will be easy enough without the Chinese Remainder optimizations. 6.3.5 Arcane RSA Threats Any number x < n is a signature of xe mod n. So it's trivial to forge someone's signature if you don't care what you're signing. The trick is to find a way to sign a specific number. Typically what is being signed is sufficiently constrained so that a random number has negligible probability of being a valid message. For example, often what is being signed is a message digest padded in a specific manner. If the pad is hundreds of zero bits, it is extremely unlikely that a random number will look like a padded message digest. If you're not careful about how you pad your message data, you may allow an attacker without knowledge of your private key to forge your signature on valid messages. Note: RSA deals with large numbers, and there is unfortunately more than one way to represent such numbers. In what follows, we have chosen to order the octets left to right from most significant to least significant. 6.3.5.1 Smooth Numbers A smooth number is defined as one that is the product of reasonably small primes. There's no absolute definition of a smooth number, since there's no real definition of reasonably small. The more compute power the attacker has at her disposal, and the more signatures she has access to, the larger the primes can be. 156 PUBLIC KEY ALGORITHMS 6.3.5.2 The threat we are about to describe is known as the smooth number threat. It is really only of theoretical interest, because of the immense amount of computation, gathering of immense numbers of signed messages, and luck involved. However, it costs very little to have an encoding that avoids this threat. The smooth number threat was discovered by Desmedt and Odlyzko [DESM86]. The first observation is that if you have signed m1 and m2, and a bad guy Carol can see your signature on m1 and m2, she can compute your signature on m1·m2, and on m1/m2, and m1 j , and on m1 j ·m2 k . For instance, if Carol sees m1 d mod n (which is your signature on m1), then she can compute your signature on m1 2 by computing (m1 d mod n) 2 mod n (see Homework Problem 8). If Carol collects a lot of your signed messages, she will be able to compute your signature on any message that can be computed from her collection by multiplication and division. If the messages you sign are mostly smooth, there will be a lot of other smooth messages on which she will be able to forge your signature. Suppose she collects your signatures on two messages whose ratio is a prime. Then she can compute your signature on that prime. If she's lucky enough to get many such message pairs, she can compute your signature on lots of primes, and then she can forge your signature on any message that is the product of any subset of those primes, each raised to any power. With enough pairs, she will be able to forge your signature on any message that is a smooth number. Actually, Carol does not have to be nearly that lucky. With as few as k signatures on messages which are products of different subsets of k distinct primes, she will be able to isolate the signatures on the individual primes through a carefully chosen set of multiplications and divisions. The typical thing being signed with RSA is a padded message digest. If it is padded with zeroes, it is much more likely to be smooth than is a random mod n number. A random mod n quantity is extremely unlikely to be smooth (low enough probability so that if you are signing random mod n numbers, we can assume Carol would have to have a lot of resources and a lot of luck to find even one smooth number you've signed, and she might need millions of them in order to mount the attack). Padding on the left with zeroes keeps the padded message digest small and therefore likely to be smooth. Padding on the right with zeroes is merely multiplying the message digest by some power of 2, and so isn't any better. Another tempting padding scheme is to pad on the right with random data. That way, since you are signing fairly random mod n numbers, it is very unlikely that any of the messages you sign will be smooth, so Carol won't have enough signed smooth messages to mount the threat. However, this leaves us

open to the next obscure threat. 6.3.5.2 The Cube Root Problem Let's say you pad on the right with random data. You chose that scheme so that there is a negligible probability that anything you sign will be smooth. However, if the public exponent is 3, this enables Carol to forge your signature on virtually any message she chooses! 6.3.6 RSA 157 Let's say Carol wants your signature on some message. The message digest of that message is h. Carol pads h on the right with zeroes. She then computes its ordinary cube root and rounds up to an integer r. Now she has forged your signature, because re = r3 = (h padded on the right with a seemingly random number). 6.3.6 Public-Key Cryptography Standard (PKCS) It is useful to have some standard for the encoding of information that will be signed or encrypted through RSA, so that different implementations can interwork, and so that the various pitfalls with RSA can be avoided. Rather than expecting every user of RSA to be sophisticated enough to know about all the attacks and develop appropriate safety measures through careful encoding, RSADSI has developed a standard known as PKCS which recommends encodings. PKCS is actually a set of standards, called PKCS #1 through PKCS #15. There are also two companion documents, An overview of the PKCS standards, and A layman's guide to a subset of ASN.1, BER, and DER. (ASN.1 = Abstract Syntax Notation 1, BER = Basic Encoding Rules, and DER = Distinguished Encoding Rules—aren't you glad you asked?). The PKCS standards define the encodings for things such as an RSA public key, an RSA private key, an RSA signature, a short RSA-encrypted message (typically a secret key), a short RSA-signed message (typically a message digest), and password-based encryption. The threats that PKCS has been designed to deal with are: • encrypting guessable messages • signing smooth numbers • multiple recipients of a message when e = 3 • encrypting messages that are less than a third the length of n when e = 3 • signing messages where the information is in the high-order part and e = 3 6.3.6.1 Encryption PKCS #1 defines a standard for formatting a message to be encrypted with RSA. RSA is not generally used to encrypt ordinary data. The most common quantity that would be encrypted with RSA is a secret key, and for performance reasons the ordinary data would be encrypted with the secret key. The recommended standard is 0 2 at least eight random nonzero octets 0 data 158 PUBLIC KEY ALGORITHMS 6.3.6.2 The actual data to be encrypted, usually a secret key, is much smaller than the modulus. If it's a DES key, it's 64 bits. If multiple DES encryption is used, then there might be two DES keys there, or 128 bits. The top octet is 0, which is a good choice because this guarantees that the message m being encrypted is smaller than the modulus n. (If m were larger than n, decryption would produce m mod n instead of m.) Note that PKCS specifies that the high-order octet (not bit!) of the modulus must be non-zero. The next octet is 2, which is the format type. The value 2 is used for a block to be encrypted. The value 1 is used for a value to be signed (see next section). Each octet of padding is chosen independently to be a random nonzero value. The reason 0 cannot be used as a padding octet is that 0 is used to delimit the padding from the data. Let's review the RSA threats and see how this encoding addresses them: • encrypting guessable messages—Since there are at least eight octets of randomly chosen padding, knowing what might appear in the data does not help the attacker who would like to guess the data, encrypt it, and compare it with the ciphertext. The attacker would have to guess the padding as well, and this is infeasible. • sending the same encrypted message to more than three recipients (assuming 3 is chosen for e)—As long as the padding is chosen independently for each recipient, the quantities being encrypted will not be the same. • encrypting messages that are less than a third the length of n when e = 3—Because the second octet is nonzero, the message will be guaranteed to be more than a third the length of n. 6.3.6.2 Encryption—Take 2 There was an attack on SSL (see Chapter 25 SSL/TLS) that could have been interpreted as a flaw in the design of SSL, but the world has come to see it as a flaw in the PKCS #1 encryption format, and there is a PKCS #1 version 2 format that fixes the "flaw". The attack is

known as the million message attack, and occurs because SSL made some incorrect assumptions about the services PKCS #1 padding provides. In the SSL protocol, the client sends the server a randomly chosen key padded according to PKCS #1 and encrypted using RSA. SSL decrypts the value, and, if the padding is correct, sends a response encrypted with the enclosed key. If after the decryption the padding is not correct, it sends an error message. The problem is that this allows an attacker to use the server as an oracle: it can send the server a message and the server will tell it whether the message (when decrypted) has proper PKCS #1 padding. Some SSL servers were particularly helpful and would say whether the padding was wrong because the first two octets were something other than 0 and 2 or whether it 6.3.6.3 RSA 159 was wrong because the length of the encrypted quantity was something other than what was expected. Daniel Bleichenbacher [BLEI98] figured out how an eavesdropper who picked up a key encrypted for a server could carefully craft variations of that encrypted key such that if the server would identify some of those variations which when decrypted began with the octets 0 and 2, the attacker could eventually figure out the encrypted key. The most helpful servers would identify one message in 216 as having that form, allowing the eavesdropper to recover the encrypted key after sending about a million messages (most of them invalid). Attacks of this sort can be avoided if the padding for encryption includes enough redundancy that the probability of a randomly chosen value decrypting into something that looks like it is properly padded is negligible. (One in a million is not considered negligible to cryptographers. To them “negligible” should be less than one in 2100 or so.) A particularly complex scheme for doing that is specified in PKCS #1 version 2, also known as OAEP [BELL94] and standardized in IEEE P1363. Because this is an obscure attack easily avoided by other means (like not being so helpful when people send you invalid messages), the world has not scrambled to migrate to this new padding. But it will probably be mandated in newly defined protocols where backwards compatibility is not an issue. 6.3.6.3 Signing PKCS #1 also defines a standard for formatting a message to be signed with RSA. Usually the data being signed is a message digest, typically 128 bits. As with encryption, padding is required. As with encryption, the top octet of 0 ensures that the quantity to be signed will be less than n. The next octet is the PKCS type, in this case, a quantity to be signed. The padding ensures that the quantity to be signed is very large and therefore unlikely to be a smooth number. Inclusion of the digest type instead of merely the digest serves two purposes. It standardizes how to tell the other party which digest function you used, and it prevents an obscure threat. The threat is that one of the message digest functions, say MD4, might be weak, so that bad guys with big budgets can generate a message with a particular MD4 message digest. Now suppose you were suspicious of MD4 and therefore used MD5. You signed the MD5 message digest of your message m. If the digest type were not included in the quantity that you RSA signed, then a bad guy could generate some message m' such that MD4(m') = MD5(m), and use the same signature you’d generated for m as the signature for m'. Including the digest type in the signature means you’re at risk only for the cryptographic strength of the message digest functions you choose to use. 0 1 at least eight octets of ff16 0 ASN.1-encoded digest type and digest 160 PUBLIC KEY ALGORITHMS 6.4 6.4 DIFFIE-HELLMAN The Diffie-Hellman public key cryptosystem predates RSA and is in fact the oldest public key system still in use. It is less general than RSA (it does neither encryption nor signatures), but it offers better performance for what it does. If Diffie-Hellman neither encrypts nor signs, how can it be called a cryptosystem? What does it do? Diffie-Hellman allows two individuals to agree on a shared key, even though they can only exchange messages in public. In other words, assume our famous two people, Alice and Bob, want to have a secret number that they share so that they can start encrypting messages to each other. But the only way they can talk is by some means of communication that lots of people can overhear. For instance,

they might be sending messages across a network, or on a telephone that might be tapped, or they might be shouting at each other across a crowded room, or they might for some reason have to resort to communication by placing ads in the personals section of the local newspaper. In its original conception, the Diffie-Hellman algorithm has limited functionality, since the only thing it really accomplishes is having a secret number that both Alice and Bob know, and nobody else can figure out based on the messages they overhear between Alice and Bob. Neither Alice nor Bob start out with any secrets, yet after the exchange of two messages that the world can overhear, Alice and Bob will know a secret number. Once they know a secret number, they can use conventional cryptography (secret key cryptography like DES, for instance) for encryption. Diffie-Hellman is actually used for key establishment (getting two things to agree on a common secret key) in some applications, for instance data link encryption on a LAN. A weakness of Diffie-Hellman is that although two individuals can agree on a shared secret key, there is no authentication, which means that Alice might be establishing a secret key with a bad guy. We will talk about this more after we force you to read through how Diffie-Hellman works. To start out, there are numbers p and g, where p is a large prime and g is a number less than p with some restrictions that aren't too important for a basic understanding of the algorithm. p and g are known beforehand and can be publicly known. For instance, Alice could choose a p and g and send them (publicly) to Bob, for instance by publishing them in The New York Times. Dear Bob, the magic number is 189028930478923478927918902890 2. Wish you were here. Love, Alice. 6.4.1 DIFFIE-HELLMAN 161 Once Alice and Bob agree on a p and g, each chooses a 512-bit number at random and keeps it secret. Let's call Alice's secret number SA and Bob's secret number SB. Each raises g to their secret number, mod p. The result is that Alice computes some number TA and Bob computes some number TB. They exchange their Ts. Finally, each raises the received T to their secret number. Nobody else can calculate gSASB in a reasonable amount of time even though they know gSA and gSB. If they could compute discrete logarithms, i.e. figure out SA based on seeing gSA, then they could figure out the Alice/Bob shared key. But we assume they can't compute discrete logarithms, because of the Fundamental Tenet of Cryptography (mathematicians haven't figured out how to do that easily in spite of considerable effort, or at least they haven't told us they have). 6.4.1 The Bucket Brigade/Man-in-the-Middle Attack If Alice receives TB indirectly, there is no way for her to know for sure whether the number came from Bob. She will establish a secret key with whoever transmitted TB, but it certainly might not be Bob. Let's assume Alice is talking to X, who may or may not be Bob. Once Alice and X establish a secret key, they can encrypt all their messages so that only Alice and X can read them. Let's say the first thing Alice and X exchange in their encrypted communication is a password that Alice and Bob have previously agreed upon, one password that Bob is to say to Alice, perhaps The fish are green, and one that Alice is to say to Bob, for instance The moon sets at midnight. If Alice receives the expected password from X, can she assume she is talking to Bob? (Think about this a Dear Bob, I'd like our prime to be 128903289023 and our g to be 23489. Love, Alice. Alice picks SA at random. Bob picks SB at random. Alice computes TA= gSA mod p. Bob computes TB = gSB mod p. They exchange Ts (in either order or simultaneously): TA ↔ TB Now each raises the number they receive to their private secret number: Alice computes TB SA mod p. Bob computes TA SB mod p. They will both come up with the same number. That is because TB SA = (gSB) SA = gSB SA = gSASB = (gSA) SB = TA SB mod p. 162 PUBLIC KEY ALGORITHMS 6.4.1 bit before reading the next paragraph—it's fun. Hint: Obviously, there must be some subtle attack or we wouldn't claim it was an interesting question.) Assume p and g are publicly known (if not, Alice can put them into her message). Alice places the ad Dear Bob. I'd like to talk to you. 8389. Love, Alice. Suppose there's a bad guy, Mr. X, who works for the newspaper. He makes one copy of the newspaper

with Alice's ad printed as Alice wished, and bribes the newspaper deliverer to give that copy to Alice. Meanwhile, Mr. X picks his own SX and computes gSX mod p. He edits the ad slightly by substituting this number instead of 8389, and has that version printed in the rest of the newspapers. Later, Bob replies, by ordering the ad So pleased to talk to you. My magic number is 9267. Love, Bob. Mr. X makes one copy of the newspaper with Bob's ad printed as Bob wished, and arranges for Bob to receive that copy. Mr. X edits Bob's ad slightly to substitute his own number for Bob's, and arranges for the newspaper with that version of the ad to get to Alice. Mr. X computes KAX = 8389SX and uses that for talking to Alice, and computes KBX = 9267SX and uses that for talking to Bob. Now suppose Alice sends an encrypted message (to Mr. X) which includes the password The fish are green. Mr. X can decrypt the message because it is encrypted using the key he shares with Alice. Mr. X reencrypts and transmits Alice's password to Bob, which reassures Bob that he is indeed talking to Alice, and Bob then transmits (encrypted, to Mr. X) The moon sets at midnight. Mr. X decrypts Bob's message, extracts the password, and reencrypts and transmits the password to Alice. Now both Alice and Bob think they are talking to each other. To guard against this threat, perhaps Alice and Bob should transmit the actual secret number they think they are using, over the encrypted channel they have established? That won't work either. Alice sends the message I think we are using KAX; however, Mr. X decrypts the message and edits it to be I think we are using KBX before encrypting and forwarding it on to Bob. Suppose instead Alice and Bob attempt to reassure themselves that they are indeed talking to each other by asking each other personal questions. If Bob asks What movie did we see the first night we met in Paris?, Mr. X merely decrypts the message with KBX, encrypts it with KAX, and forwards the message on to Alice. When Alice replies, Mr. X merely forwards the message on to Alice Mr. X Bob gSA = 8389 gSX = 5876 gSB = 9267 8389 5876 5876 9267 shared key KAX shared key KBX 5876SA = 8389SX 9267SX = 5876SB Figure 6-4. Bucket Brigade/Man-in-the-Middle Attack 6.4.2 DIFFIE-HELLMAN 163 Bob. (We're assuming Alices's memory of past evenings makes it worthwhile for Mr. X to actually wait for her answer rather than making one up himself.) The name bucket brigade attack comes from the way firefighters of old formed a line of people between a water source and a fire and passed full buckets toward the fire and empty buckets back. But the less politically correct term man-in-the-middle has become more common. After establishing the shared keys, Mr. X passes messages back and forth and can examine them and/or modify them as they go. Using Diffie-Hellman alone, there's really nothing Alice and Bob can do to detect the intruder through whom they are communicating. As a result, this form of Diffie-Hellman is only secure against passive attack where the intruder just watches the messages. 6.4.2 Defenses Against Man-in-the-Middle Attack 6.4.2.1 Published Diffie-Hellman Numbers One technique by which Diffie-Hellman can be secure against active attacks is for each person to have a somewhat permanent public and secret number instead of inventing one for each exchange. For this to work, everyone (in the communicating set) has to agree on a common p and g. The public numbers are then all published by some means that is assumed reliable (for instance, through a PKI, see Chapter 13 PKI (Public Key Infrastructure)). To the extent an intruder can't get in and modify the published public numbers, this makes Diffie-Hellman immune to active attacks. It has the additional advantage of eliminating the first two messages of the protocol. Knowing my own secret and looking up the public number of the person with whom I want to communicate, I can compute a key that the two of us will share for all messages we send to one another. 6.4.2.2 Authenticated Diffie-Hellman If Alice and Bob know some sort of secret with which they can authenticate each other, either a shared secret key or knowledge of each other's public keys (and their own private keys), then they can use this secret to prove that it was they who generated their Diffie-Hellman values. We call such an exchange an authenticated Diffie-

Hellman exchange. The proof can be done simultaneously with sending the Diffie-Hellman value, or after the Diffie-Hellman exchange. Examples are: • Encrypt the Diffie-Hellman exchange with the pre-shared secret. • Encrypt the Diffie-Hellman value with the other side's public key (see Homework Problem 2). • Sign the Diffie-Hellman value with your private key. 164 PUBLIC KEY ALGORITHMS 6.4.3 • Following the Diffie-Hellman exchange, transmit a hash of the agreed-upon shared Diffie-Hellman value, your name, and the pre-shared secret. • Following the Diffie-Hellman exchange, transmit a hash of the pre-shared secret and the Diffie-Hellman value you transmitted. 6.4.3 Encryption with Diffie-Hellman We've described the lack of authentication with Diffie-Hellman. There's another disadvantage with classic Diffie-Hellman: in order for two individuals to communicate, they have to first have an active exchange. This is easy to remedy. Suppose Alice is working late and wants to send an encrypted message to Bob that Bob will be able to read when he shows up at work the next morning at 7 AM (and Alice has no intention of being around at 7 AM). First everyone computes a public key, which consists of the three numbers ⟨p, g, T⟩, where T = gS mod p, for the private key S. These public keys are displayed in a reliable and public place (like being published in The New York Times). So Bob has published a ⟨pB, gB, TB⟩. If Alice wants to send Bob an encrypted message, she picks a random number SA, computes gB SA mod pB and computes KAB = TB SA mod pB, and uses that as the encryption key to share with Bob. She uses KAB to encrypt the message according to any secret key cryptographic technique, and sends the encrypted message, along with gB SA mod pB to Bob. Bob raises gB SA mod pB to his own secret SB, and thereby calculates KAB which enables him to decrypt the message. 6.4.4 ElGamal Signatures Using the same sort of keys as Diffie-Hellman, ElGamal came up with a signature scheme [EFF98]. It is much harder to understand than signing with RSA. While it's important to know that it's possible and to understand the political and performance implications, the mathematics is tedious and unintuitive. We reluctantly recommend that all but true die-hards skip this section. ElGamal signatures require each individual to have a long-term public/private key pair (the public key being ⟨g, p, T⟩ and the secret key being S, where gS mod p = T, as described for Diffie-Hellman), and (surprisingly) require an individual to generate a new and different public/private key pair for each item that needs to be signed. Luckily the per-message public/private key pair is easy to compute. For a particular message m, choose a random number Sm and (using the same g and p as in the long-term key) compute gSm mod p = Tm. To use ElGamal, there has to be a message digest function that is well-known. Given a message m, to compute a signature, you first compute the message digest of m|Tm. Call that message digest dm. Then you calculate Sm + dmS mod (p−1), 6.4.5 DIFFIE-HELLMAN 165 which is the signature. Let's call that X (since we've already used S for the Secret number, and some people sign documents with an X). m is transmitted, along with X and Tm. To verify this signature, you compute dm and check that gX = TmTdm mod p. This will be true, assuming the signature is valid, because gX = gSm+dmS = gSmgdmS = TmTdm mod p. This is not terrifically intuitive. The important things to try to convince oneself of are: • If the signature is done correctly, the verification will succeed. • If the message is modified after being signed, the inputs to the signature function will have changed and with overwhelming probability the signature will not match the modified message. • Knowledge of the signature will not help divulge the signer's private key, S. • Someone without knowledge of S will not be able to produce a valid signature. 6.4.5 Diffie-Hellman Details—Safe Primes While Diffie-Hellman works with any prime p and any number g, it is less secure if p and g don't have additional mathematical properties. It turns out that for obscure mathematical reasons it is particularly nice if (p−1)/2 is also prime. A prime p that satisfies this additional constraint is called a safe prime, or a Sophie Germain prime. It is also particularly nice if gx ≠ 1 mod p unless x = 0 mod p−1. If p is a safe prime, this is satisfied by any g ≠ −1 mod p for which

g(p−1)/2 = −1 mod p, which is true for almost half of all mod p numbers. It is computationally expensive to choose p and g. Theoretically, one only needs to do it once. You could keep using the same p and g. It could even be a standard, and everyone could use the same p and g. However, that is not advisable. It turns out to be possible, though incredibly spaceand computation-intensive, to calculate a large table based on a single p, which would allow you to compute discrete logarithms for that p. A similar scheme would allow you to break RSA, in the sense of being able to calculate someone's private key based on their public key, but there is not that much incentive to do so because that would only allow you to break one person's key. If the p for Diffie-Hellman were broken, then every key exchange based on Diffie-Hellman could be broken. Simply changing p occasionally will eliminate this threat. 166 PUBLIC KEY ALGORITHMS 6.5 6.5 DIGITAL SIGNATURE STANDARD (DSS) NIST, the (U.S.) National Institute of Standards and Technology, has proposed an algorithm for digital signatures based on ElGamal. The algorithm is known as DSA, for Digital Signature Algorithm. As a proposed standard it is known as DSS. The differences between it and ElGamal mainly have to do with performance. Instead of all calculations being done mod p (where p is a 512-bit prime), some are done mod q (where q is a 160-bit prime which divides p−1). This makes the exponents 160 bits rather than 512 bits, which makes signing three times faster. However, there are other differences that make DSS slower than it might have been. In particular, an inverse calculation is required by both the signer and the verifier. DSS could easily have been defined to require only the signer to calculate an inverse. Instead, DSS allows the signer to precalculate its inverse before it even has a message, at the expense of requiring the verifier to also calculate an inverse. But why should we make it more convenient for the signer, given that it's less convenient for the verifier? Presumably there is at least one verification for each signature—otherwise why would one bother signing anything? There is one important application in which DSS's approach might be an important optimization: smart cards (see §8.8 Authentication Tokens). A smart card is likely to have a very low-performance processor. It will have to do a signature in order for a user to successfully log in. If the operation of signing takes a long time, say two minutes or more, it will annoy the human who is trying to use the network. Taking inverses is one of the most computationally expensive parts of generating a DSS signature. If it had to be done on a smart card at the time the user logged in, the user would be forced to wait many seconds. However, if the smart card generated some ⟨Sm, Sm −1 ⟩ pairs while the user was logged in, then when the user next wants to log in, the smart card will have already done the requisite inverse operation. The machines doing verifications, the DSS designers believe, will be fast machines, so doing an extra inverse operation will not be annoying. 6.5.1 The DSS Algorithm • Generate p and q (which will be public). Find 160-bit prime q. Find a 512-bit prime p of the form kq+1. This is a very expensive operation, but it need not be done often. As a matter of fact it can be done once and the p can be published in the standard. Everyone can use the same p, with some caveats. It is rumored to be possible for someone to generate a p in such a way that someone can break cryptographic 6.5.1 DIGITAL SIGNATURE STANDARD (DSS) 167 operations based on that p, but no other someone would be able to tell that there was anything wrong with using that p. So having a particular p published in the standard as the one everyone should use is controversial. In addition, if there is a p that everyone uses, there is an efficiency gain (because you don't have to generate new ps and qs), but the chosen p presents a target for attacking the algorithm. • Generate g (which will be public). Find a number g such that gq = 1 mod p. This is done by taking any random number h > 1 and raising it to (p−1)/q to get g. Then, since g = h(p−1)/q, gq = hp−1 = 1 by Fermat's theorem. (Actually, h can't be just any old random number. It is important that g not be 1, or else things will not be secure at all. So a new h is tried if g = h(p−1)/q turns out to be 1.) • Choose a long-term public/private key pair ⟨T, S⟩. This is done

by choosing a random S < q and setting T = gS mod p. • Choose a per message public/private key pair ⟨Tm, Sm⟩. This is done by choosing a random Sm and setting Tm = ((gSm mod p) mod q). While you're at it, calculate Sm −1 mod q so it won't need to be done in real time when signing the message. • Calculate a message digest dm of the message. The DSS has a sister function, SHS, that is a hashing algorithm recommended by NIST for use with DSS. SHS happens to hash to 160 bits, but it is only coincidence that the size of the hash matches the size of q. 160 bits seemed a logical number for both because it's about the right size and it's a multiple of 32, so it's conveniently stored on machines with 32-bit words. DSS could be used with any hash function, though no security would be gained by using one longer than 160 bits. • Compute the signature X = Sm −1(dm + STm) mod q. • Transmit all relevant information: the message m; the per-message public number Tm; the signature X. The public key information, consisting of T, p, q, and g, is known beforehand and doesn't need to be transmitted with the message. • Verify the signature: 168 PUBLIC KEY ALGORITHMS 6.5.2 Calculate the mod q inverse of the signature, X−1 . Calculate dm. Calculate x = dm· X−1 mod q. Calculate y = Tm· X−1 mod q. Calculate z = (gx · Ty mod p) mod q. If z = Tm, then the signature is verified. 6.5.2 Why Does the Verification Procedure Work? Let v = (dm + STm) −1 mod q. Then, X−1 = (Sm −1(dm + STm))−1 = Sm(dm + STm) −1 = Smv mod q, x = dm· X−1 = dmSmv mod q, y = Tm· X−1 = TmSmv mod q, z = gx · Ty = gdmSmv gSTmSmv = g(dm+STm)Smv = gSm = Tm mod p mod q. (We're not worried about the mod q in the exponents of g since gq = 1 mod p.) 6.5.3 Why Is This Secure? What does it mean to be secure? It means several things. • Signing something does not divulge the private key S. • Nobody should be able to generate a signature for a given message without knowing S. • Nobody should be able to generate a message that matches a given signature. • Nobody should be able to modify a signed message in a way that keeps the same signature valid. Why does DSS have all these properties? The Fundamental Tenet of Cryptography (nobody knows how to break it). DSS also has the blessing of the NSA, arguably the best cryptographers in the world. Unfortunately it's a mixed blessing, since some cynics believe NSA would never propose an algorithm it couldn't break. 6.5.4 DIGITAL SIGNATURE STANDARD (DSS) 169 6.5.4 The DSS Controversy DSS was published by NIST on August 30, 1991 as a proposed standard for digital signatures. NIST announced a 90-day comment period, which sparked a flurry of debate and an extension of the comment period. At the time of this writing the debate continues with no end in sight. The hidden (or sometimes not hidden) question in the debate over DSS is why NIST chose a variant of ElGamal rather than standardizing on RSA, which had become the de facto industry standard. The arguments about standardizing on DSS are: • Test of time—DSS may have undetected flaws, since it has not be subjected to the intense scrutiny of RSA. • Mandated 512-bit/160-bit moduli—DSS fixes the key length at 512 bits for p and 160 bits for q. It is true that a DSS key of 512 bits might be a little more secure than a 512-bit RSA key, because the RSA key is a composite number and the DSS key is a prime, but it certainly won't be as secure as a 600-bit RSA key. According to calculations made by Ron Rivest [RIVE91b], a determined and well-financed attacker, say with a budget of $25 million, could break DSS with a 512-bit modulus in a year. [NIST has responded by allowing key lengths up to 1024 bits for p.] • Since choosing a DSS ⟨p, q, g⟩ triple is computationally expensive, it is likely that many people will base their own key on parameters that have been published. With RSA, an attacker that breaks a key breaks only a single key. With DSS, an attacker that breaks a ⟨p, q, g⟩ triple breaks all keys upon which it is based. • Trapdoor primes—Another problem with using a published ⟨p, q, g⟩ triple is that you have to trust the source. It is possible for the source to generate a triple that is "pre-broken" in the sense that the source knows how to forge signatures for any key based on that triple. It's been claimed that the sample ⟨p, q, g⟩ published in the DSS document is suspicious, in that it has interesting