organizations may have discovered this technology earlier). Unlike secret key cryptography, keys are not shared. Instead, each individual has two keys: a private key that need not be revealed to anyone, and a public key that is preferably known to the entire world. Note that we call the private key a private key and not a secret key. This convention is an attempt to make it clear in any context whether public key cryptography or secret key cryptography is being used. Some people use the term secret key for the private key in public key cryptography, or use the term private key for the secret key in secret key technology. We hope to convince people to use the term secret key only as the single secret number used in secret key cryptography. The term private key should refer to the key in public key cryptography that must not be made public. Unfortunately, both words public and private begin with p. We will sometimes want a single letter to refer to one of the keys. The letter p won't do. We will use the letter e to refer to the public key, since the public key is used when encrypting a message. We'll use the letter d to refer to the 2.5 PUBLIC KEY CRYPTOGRAPHY 45 private key, because the private key is used to decrypt a message. Encryption and decryption are two mathematical functions that are inverses of each other. There is an additional thing one can do with public key technology, which is to generate a digital signature on a message. A digital signature is a number associated with a message, like a checksum or the MAC described in §2.4.5 Integrity Check. However, unlike a checksum, which can be generated by anyone, a digital signature can only be generated by someone knowing the private key. A public key signature differs from a secret key MAC because verification of a MAC requires knowledge of the same secret as was used to create it. Therefore anyone who can verify a MAC can also generate one, and so be able to substitute a different message and corresponding MAC. In contrast, verification of the signature only requires knowledge of the public key. So Alice can sign a message by generating a signature only she can generate, and other people can verify that it is Alice's signature, but cannot forge her signature. This is called a signature because it shares with handwritten signatures the property that it is possible to recognize a signature as authentic without being able to forge it. plaintext ciphertext encryption public key ciphertext plaintext decryption private key plaintext signed message signing private key signed message plaintext verification public key 46 INTRODUCTION TO CRYPTOGRAPHY 2.5.1 2.5.1 Security Uses of Public Key Cryptography Public key cryptography can do anything secret key cryptography can do, but the known public key cryptographic algorithms are orders of magnitude slower than the best known secret key cryptographic algorithms and so are usually used together with secret key algorithms. Public key cryptography is very useful because network security based on public key technology tends to be more easily configurable. Public key cryptography might be used in the beginning of communication for authentication and to establish a temporary shared secret key, then the secret key is used to encrypt the remainder of the conversation using secret key technology. For instance, suppose Alice wants to talk to Bob. She uses his public key to encrypt a secret key, then uses that secret key to encrypt whatever else she wants to send him. Since the secret key is much smaller than the message, using the slow public key cryptography to encrypt the secret key is not that much of a performance hit. Only Bob can decrypt the secret key. He can then communicate using that secret key with whoever sent that message. Notice that given this protocol, Bob does not know that it was Alice who sent the message. This could be fixed by having Alice digitally sign the encrypted secret key using her private key. Now we'll describe the types of things one might do with public key cryptography. 2.5.2 Transmitting Over an Insecure Channel Suppose Alice's ⟨public key, private key⟩ pair is ⟨eA, dA⟩. Suppose Bob's key pair is ⟨eB, dB⟩. Assume Alice knows Bob's public key, and Bob knows Alice's public key. Actually, accurately learning other people's public keys is one of the biggest challenges in using public key cryptography and will be discussed in detail in Chapter 13 PKI (Public Key

Infrastructure). But for now, don't worry about it. 2.5.3 Secure Storage on Insecure Media This is really the same as what one would do with secret key cryptography. You'd encrypt the data with your public key. Then nobody can decrypt it except you, since decryption will require the use of the private key. For performance reasons, you probably wouldn't encrypt the data directly with the public key, but rather randomly generate a secret key, encrypt the data with that secret key, and encrypt that secret key with the public key. As with secret key technology, if you lose your private Alice Bob encrypt mA using eB decrypt to mA using dB decrypt to mB using dA encrypt mB using eA 2.5.4 PUBLIC KEY CRYPTOGRAPHY 47 key, the data is irretrievably lost. If you are worried about that, you can encrypt an additional copy of the data encryption key under the public key of someone you trust, like your lawyer. Or you can store copies of your private key with someone you trust (see §21.9.1 Key Escrow). Public key technology has an important advantage over secret key technology for this application. Alice can encrypt a message for Bob without knowing his decryption key. 2.5.4 Authentication With secret key cryptography, if Alice and Bob want to communicate, they have to share a secret. If Bob wants to be able to prove his identity to lots of entities, then with secret key technology he will need to remember lots of secret keys, one for each entity to which he would like to prove his identity. Possibly he could use the same shared secret with Alice as with Carol, but that has the disadvantage that then Carol and Alice could impersonate Bob to each other. Public key technology is much more convenient. Bob only needs to remember a single secret, his own private key. It is true that if Bob wants to be able to verify the identity of thousands of entities, then he will need to know (or be able to obtain when necessary) thousands of public keys. In Chapter 13 PKI (Public Key Infrastructure) we discuss how this might be done. Here's an example of how Alice can use public key cryptography for verifying Bob's identity assuming Alice knows Bob's public key. Alice chooses a random number r, encrypts it using Bob's public key eB, and sends the result to Bob. Bob proves he knows dB by decrypting the message and sending r back to Alice. Another advantage of public key authentication is that Alice does not need to keep any secret information in order to verify Bob. For instance, Alice might be a computer system in which backup tapes are unencrypted and easily stolen. With secret key based authentication, if Carol stole a backup tape and read the key that Alice shares with Bob, she could then trick Bob into thinking she was Alice (or trick Alice into thinking she was Bob). In contrast, with public key based authentication, the only information on Alice's backup tapes is public key information, and that cannot be used to impersonate Bob. Alice Bob encrypt r using eB decrypt to r using dB r 48 INTRODUCTION TO CRYPTOGRAPHY 2.5.5 2.5.5 Digital Signatures Forged in USA engraved on a screwdriver claiming to be of brand Craftsman It is often useful to prove that a message was generated by a particular individual. This is easy with public key technology. Bob's signature for a message m can only be generated by someone with knowledge of Bob's private key. And the signature depends on the contents of m. If m is modified in any way, the signature no longer matches. So digital signatures provide two important functions. They prove who generated the information, and they prove that the information has not been modified in any way by anyone since the message and matching signature were generated. Digital signatures offer an important advantage over secret key based cryptographic MACs— non-repudiation. Suppose Bob sells widgets and Alice routinely buys them. Alice and Bob might agree that rather than placing orders through the mail with signed purchase orders, Alice will send electronic mail messages to order widgets. To protect against someone forging orders and causing Bob to manufacture more widgets than Alice actually needs, Alice will include a message integrity code on her messages. This could be either a secret key based MAC or a public key based signature. But suppose sometime after Alice places a big order, she changes her mind (the bottom fell out of the widget market). Since there's a big penalty for canceling an order, she doesn’t fess up that she’s canceling, but instead denies that she ever placed the order. Bob sues. If Alice authenticated the message by computing a MAC based on a key she shares with Bob, Bob knows Alice really placed the order because nobody other than Bob and Alice know that key. If Bob knows he didn’t create the message he knows it must have been Alice. But he can’t prove it to anyone! Since he knows the same secret key that Alice used to sign the order, he could have forged the signature on the message himself and he can’t prove to the judge that he didn’t! If it was a public key signature, he can show the signed message to the judge and the judge can verify that it was signed with Alice’s key. Alice can still claim of course that someone must have stolen and misused her key (it might even be true!), but the contract between Alice and Bob could reasonably hold her responsible for damages caused by her inadequately protecting her key. Unlike secret key cryptography, where the keys are shared, you can always tell who’s responsible for a signature generated with a private key. 2.6 HASH ALGORITHMS Hash algorithms are also known as message digests or one-way transformations. A cryptographic hash function is a mathematical transformation that takes a message of arbitrary length (transformed into a string of bits) and computes from it a fixed-length (short) number. 2.6.1 HASH ALGORITHMS 49 We’ll call the hash of a message m, h(m). It has the following properties: • For any message m, it is relatively easy to compute h(m). This just means that in order to be practical it can’t take a lot of processing time to compute the hash. • Given h(m), there is no way to find an m that hashes to h(m) in a way that is substantially easier than going through all possible values of m and computing h(m) for each one. • Even though it’s obvious that many different values of m will be transformed to the same value h(m) (because there are many more possible values of m), it is computationally infeasible to find two values that hash to the same thing. An example of the sort of function that might work is taking the message m, treating it as a number, adding some large constant, squaring it, and taking the middle n digits as the hash. You can see that while this would not be difficult to compute, it’s not obvious how you could find a message that would produce a particular hash, or how one might find two messages with the same hash. It turns out this is not a particularly good message digest function—we’ll give examples of secure message digest functions in Chapter 5 Hashes and Message Digests. But the basic idea of a message digest function is that the input is mangled so badly the process cannot be reversed. 2.6.1 Password Hashing When a user types a password, the system has to be able to determine whether the user got it right. If the system stores the passwords unencrypted, then anyone with access to the system storage or backup tapes can steal the passwords. Luckily, it is not necessary for the system to know a password in order to verify its correctness. (A proper password is like pornography. You can’t tell what it is, but you know it when you see it.) Instead of storing the password, the system can store a hash of the password. When a password is supplied, it computes the password’s hash and compares it with the stored value. If they match, the password is deemed correct. If the hashed password file is obtained by an attacker, it is not immediately useful because the passwords can’t be derived from the hashes. Historically, some systems made the password file publicly readable, an expression of confidence in the security of the hash. Even if there are no cryptographic flaws in the hash, it is possible to guess passwords and hash them to see if they match. If a user is careless and chooses a password that is guessable (say, a word that would appear in a 50000-word dictionary or book of common names), an exhaustive search would “crack” the password even if the encryption were sound. For this reason, many systems hide the hashed password list (and those that don’t should). 50 INTRODUCTION TO CRYPTOGRAPHY 2.6.2 2.6.2 Message Integrity Cryptographic hash functions can be used to generate a MAC to protect the integrity of messages transmitted over insecure media in much the same way as secret key cryptography. If we merely sent the message and used the hash of the message as a MAC, this would not be secure, since the hash function is well-known. The bad guy can modify the message and compute a new hash for the new message, and transmit that. However, if Alice and Bob have agreed on a secret, Alice can use a hash to generate a MAC for a message to Bob by taking the message, concatenating the secret, and computing the hash of message|secret. This is called a keyed hash. Alice then sends the hash and the message (without the secret) to Bob. Bob concatenates the secret to the received message and computes the hash of the result. If that matches the received hash, Bob can have confidence the message was sent by someone knowing the secret. [Note: there are some cryptographic subtleties to making this actually secure; see §5.2.2 Computing a MAC with a Hash]. 2.6.3 Message Fingerprint If you want to know whether some large data structure (e.g. a program) has been modified from one day to the next, you could keep a copy of the data on some tamper-proof backing store and periodically compare it to the active version. With a hash function, you can save storage: you simply save the message digest of the data on the tamper-proof backing store (which because the hash is small could be a piece of paper in a filing cabinet). If the message digest hasn't changed, you can be confident none of the data has. A note to would-be users—if it hasn't already occurred to you, it has occurred to the bad guys—the program that computes the hash must also be independently protected for this to be secure. Otherwise the bad guys can change the file but also change the hashing program to report the checksum as though the file were unchanged! message secret hash hash = ? secret Alice Bob 2.6.4 HOMEWORK 51 2.6.4 Downline Load Security It is common practice to have special-purpose devices connected to a network, like routers or printers, that do not have the nonvolatile memory to store the programs they normally run. Instead, they keep a bootstrap program smart enough to get a program from the network and run it. This scheme is called downline load. Suppose you want to downline load a program and make sure it hasn't been corrupted (whether intentionally or not). If you know the proper hash of the program, you can compute the hash of the loaded program and make sure it has the proper value before running the program. 2.6.5 Digital Signature Efficiency The best-known public key algorithms are sufficiently processor-intensive that it is desirable to compute a message digest of the message and sign that, rather than to sign the message directly. The message digest algorithms are much less processor-intensive, and the message digest is much shorter than the message. 2.7 HOMEWORK 1. What is the dedication to this book? 2. Random J. Protocol-Designer has been told to design a scheme to prevent messages from being modified by an intruder. Random J. decides to append to each message a hash of that message. Why doesn't this solve the problem? (We know of a protocol that uses this technique in an attempt to gain security.) 3. Suppose Alice, Bob, and Carol want to use secret key technology to authenticate each other. If they all used the same secret key K, then Bob could impersonate Carol to Alice (actually any of the three can impersonate the other to the third). Suppose instead that each had their own secret key, so Alice uses KA, Bob uses KB, and Carol uses KC. This means that each one, to prove his or her identity, responds to a challenge with a function of his or her secret key and the challenge. Is this more secure than having them all use the same secret key K? (Hint: what does Alice need to know in order to verify Carol's answer to Alice's challenge?) 4. As described in §2.6.4 Downline Load Security, it is common, for performance reasons, to sign a message digest of a message rather than the message itself. Why is it so important that it be difficult to find two messages with the same message digest? 52 INTRODUCTION TO CRYPTOGRAPHY 2.7 5. What's wrong with the protocol in §2.4.4 Authentication? (Hint: assume Alice can open two connections to Bob.) 6. Assume a cryptographic algorithm in which the performance for the good guys (the ones that know the key) grows linearly with the length of the key, and for which the only way to break it is a brute-force attack of trying all possible keys. Suppose the performance for the good guys is adequate (e.g., it can encrypt and decrypt as fast as the bits can be transmitted over the wire) at a certain size key. Then suppose advances in computer technology make computers twice as fast. Given that both the good guys and the bad guys get faster computers, does this advance in computer speed work to the advantage of the good guys, the bad guys, or does it not make any difference? 53 3 SECRET KEY CRYPTOGRAPHY 3.1 INTRODUCTION This chapter describes how secret key cryptographic algorithms work. It describes in detail the DES and IDEA algorithms. These algorithms take a fixed-length block of message (64 bits in the case of both DES and IDEA), a fixed-length key (56 bits for DES and 128 bits for IDEA) and generate a block of output (the same length as the input). In general, a message won't happen to be 64 bits long. In §4.2 Encrypting a Large Message we'll discuss how to convert the basic fixed-length block encryption algorithm into a general message encryption algorithm. 3.2 GENERIC BLOCK ENCRYPTION A cryptographic algorithm converts a plaintext block into an encrypted block. It's fairly obvious that if the key length is too short (for instance, 4 bits), the cryptographic scheme would not be secure because it would be too easy to search through all possible keys. There's a similar issue with the length of the block of plaintext to be encrypted. If the block length is too short (say one octet, as in a monoalphabetic cipher), then if you ever had some paired ⟨plaintext, ciphertext⟩, you could construct a table to be used for decryption. It might be possible to obtain such pairs because messages might only remain secret for a short time, perhaps because the message says where the army will attack the next day. Having a block length too long is merely inconvenient—unnecessarily complex and possibly having performance penalties. 64 bits is a reasonable length, in that you are unlikely to get that many blocks of ⟨plaintext,ciphertext⟩ pairs, and even if you did, it would take too much space to store the table (264 entries of 64 bits each) or too much time to sort it for efficient searching. The most general way of encrypting a 64-bit block is to take each of the 264 input values and map it to a unique one of the 264 output values. (It is necessary that the mapping be one-to-one, i.e. 54 SECRET KEY CRYPTOGRAPHY 3.2 only one input value maps to any given output value, since otherwise decryption would not be possible.) Suppose Alice and Bob (who happen to speak a language in which all sentences are 64 bits long) want to decide upon a mapping that they can use for encrypting their conversations. How would they specify one? To specify a monoalphabetic cipher with English letters takes 26 specifications of 26 possible values, approximately. For instance, How would you specify a mapping of all possible 64-bit input values? Well, let's start: Hmm, we probably don't want to write this all out. There are 264 possible input values and for each one we have to specify a 64-bit output value. This would take 270 bits. (Actually, nitpickers might note that there aren't quite 270 bits of information since the mapping has to be a permutation, i.e., each output value is used exactly once, so for instance the final output value does not need to be explicitly specified—it's the one that's left over. However, there are 264! different possible permutations of 264 values, which would take more than 269 bits to represent.) [Remember n! (read "n factorial") is n·(n−1)·(n−2)·(n−3)···3·2·1. It can be approximated by Stirling's formula: ] So let's say it would take 269 bits to specify the mapping. That 269 bit number would act like a secret key that Alice and Bob would share. But it is doubtful that they could remember a key that large, or even be able to say it to each other within a lifetime, or store it on anything. So this is not particularly practical. Secret key cryptographic systems are designed to take a reasonable-length key (i.e., more like 64 bits than 264 bits) and generate a one-to-one mapping that looks, to someone who does not know the key, completely random. Random means that it should look, to someone who doesn't know the key, as if the mapping from an input value to an output value were generated by using a random number generator. (To get the mapping from input i to output o, flip a 264-sided coin to choose the value of o—or if such a coin is not readily available, a single coin could be flipped 64 times. Since the mapping must be one-to-one, you'll have to start over again choosing o if the value selected by the coin has been previously used.) If the mapping were truly random, any single bit change to the input will result in a totally independently chosen random number output. The two different output numbers should have no correlation, meaning that about half the bits should be the same and about half the bits should be different. For instance, it can't be the case that the 3rd bit of output always changes if the 12th bit of input changes. So the cryptographic algorithms are designed to spread bits a→q b→d c→w d→x e→a f→f g→z h→b etc. 0000000000000000→8ad1482703f217ce 0000000000000001→b33dc8710928d701 0000000000000002→29e856b28013fa4c n ne n n n ! ≈ – 2π 3.2 GENERIC BLOCK ENCRYPTION 55 around, in the sense that a single input bit should have influence on all the bits of the output, and be able to change any one of them with a probability of about 50% (depending on the values of the other 63 bits of input). There are two kinds of simple transformations one might imagine on a block of data, and they are named in the literature as substitutions and permutations (which is the only reason we are using those names—we would have chosen different words, perhaps the term bit shuffle instead of permutation). Let's assume we are encrypting k-bit blocks: A substitution specifies, for each of the 2k possible values of the input, the k-bit output. As noted above, this would be impractical to build for 64-bit blocks, but would be possible with blocks of length, say, 8 bits. To specify a completely randomly chosen substitution for k-bit blocks would take about k·2k bits. A permutation specifies, for each of the k input bits, the output position to which it goes. For instance, the 1st bit might become the 13th bit of output, the 2nd bit would become the 61st bit of output, and so on. To specify a completely randomly chosen permutation of k bits would take about k log2k bits (for each of the k bits, one has to specify which bit position it will be in the output, and it only takes log2k bits to specify k values). A permutation is a special case of a substitution in which each bit of the output gets its value from exactly one of the bits of the input. The number of permutations is sufficiently small that it is possible to specify and build an arbitrary 64-bit permuter. One possible way to build a secret key algorithm is to break the input into manageable-sized chunks (say 8 bits), do a substitution on each small chunk, and then take the outputs of all the substitutions and run them through a permuter that is as big as the input, which shuffles the bits around. Then the process is repeated, so that each bit winds up as input to each of the substitutions. (See Figure 3-1.) Each time through is known as a round. If we do only a single round, then a bit of input can only affect 8 bits of output, since each input bit goes into only one of the substitutions. On the second round, the 8 bits affected by a particular input bit get spread around due to the permutation, and assuming each of those 8 bits goes into a different substitution, then the single input bit will affect all the output bits. Just as when shuffling a deck of cards, there is an optimal number of rounds (shuffles). Once the cards are sufficiently randomized, extra shuffles just waste time. Part of the design of an algorithm is determining the best number of rounds (for optimal security, at least enough rounds to randomize as much as possible; for efficiency reasons no more rounds than necessary). Another important feature of an encryption mechanism is it must be efficient to reverse, given the key. An algorithm like the one above would take the same effort to decrypt as to encrypt, since each of the steps can be run as efficiently backwards as forwards. 56 SECRET KEY CRYPTOGRAPHY 3.3 3.3 DATA ENCRYPTION STANDARD (DES) DES was published in 1977 by the National Bureau of Standards (since renamed to the National Institute of Standards and Technology) for use in commercial and unclassified (hmm...) U.S. Government applications. It was designed by IBM based on their own Lucifer cipher and input from NSA. DES uses a 56-bit key, and maps a 64-bit input block into a 64-bit output block. The key actually looks like a 64-bit quantity, but one bit in each of the 8 octets is used for odd parity on each octet. Therefore, only 7 of the bits in each octet are actually meaningful as a key. DES is efficient to implement in hardware but relatively slow if implemented in software. Although making software implementations difficult was not a documented goal of DES, people have asserted that DES was specifically designed with this in mind, perhaps because this would limit its use to organizations that could afford hardware-based solutions, or perhaps because it made it easier to control access to the technology. At any rate, advances in CPUs have made it feasible to do DES in software. For instance, a 500-MIP CPU can encrypt at about 30 Koctets per second (and perhaps more depending on the details of the CPU design and the cleverness of the implementation). This is adequate for many applications. 64-bit input 8 bits 8 bits 8 bits 8 bits 8 bits 8 bits 8 bits 8 bits S1 S2 S3 S4 S5 S6 S7 S8 8 bits 8 bits 8 bits 8 bits 8 bits 8 bits 8 bits 8 bits 64-bit intermediate 64-bit output Figure 3-1. Example of Block Encryption Loop for n rounds Eight 8-bit substitution functions derived from the key Divide input into eight 8-bit pieces Permute the bits, possibly based on the key 3.3 DATA ENCRYPTION STANDARD (DES) 57 Advances in semiconductor technology make the key-length issue more critical. Chip speeds have caught up so that DES keys can be broken with a bit of cleverness and exhaustive search. Perhaps a 64-bit key might have extended its useful lifetime by a few years. Given hardware price/performance improving about 40% per year, keys must grow by about 1 bit every 2 years. Assuming 56 bits was just sufficient in 1979 (when DES was standardized), 64 bits was about right in 1995, and 128 bits would suffice until 2123. Why 56 bits? Use of a 56-bit key is one of the most controversial aspects of DES. Even before DES was adopted, people outside of the intelligence community complained that 56 bits provided inadequate security [DENN82, DIFF76a, DIFF77, HELL79]. So why were only 56 of the 64 bits of a DES key used in the algorithm? The disadvantage of using 8 bits of the key for parity checking is that it makes DES considerably less secure (256 times less secure against exhaustive search). OK, so what is the advantage of using 8 bits of the key for parity? Well, uh, let's say you receive a key electronically, and you want to sanity-check it to see if it could actually be a key. If you check the parity of the quantity, and it winds up not having the correct parity, then you'll know something went wrong. There are two problems with this reasoning. One is that there is a 1 in 256 chance (given the parity scheme) that even if you were to get 64 bits of garbage, that the result will happen to have the correct parity and therefore look like a key. That is way too large a possibility of error for it to afford any useful protection to any application. The other problem with the reasoning is that there is nothing terribly bad about getting a bad key. You'll discover the key is bad when you try to use it for encrypting or decrypting. The key, at 56 bits, is pretty much universally acknowledged to be too small to be secure. Perhaps one might argue that a smaller key is an advantage because it saves storage— but that argument doesn't hold since nobody does data compression on the 64-bit keys in order to fit them into 56 bits. So what benefits are there to usurping 8 bits for parity that offset the loss in security? People (not us, surely!) have suggested that our government consciously decided to weaken the security of DES just enough so that NSA would be able to break it. We would like to think there is an alternative explanation, but we have never heard a plausible one proposed. 58 SECRET KEY CRYPTOGRAPHY 3.3.1 3.3.1 DES Overview DES is quite understandable, and has some very elegant tricks. Let's start with the basic structure of DES (Figure 3-2). How secure is DES? Suppose you have a single block of ⟨plaintext, ciphertext⟩. Breaking DES in this case would mean finding a key that maps that plaintext to that ciphertext. With DES implemented in software, it would take on the order of half a million MIP-years, through brute force, to find the key. (Is it possible to find the "wrong" key, given a particular pair? Might two different keys map the same plaintext to the same ciphertext? How many keys on the average map a particular pair? See Homework Problem 3.) Often the attacker does not have a ⟨plaintext, ciphertext⟩ block. Instead the attacker has a reasonable amount of ciphertext only. It might be known, for example, that the encrypted data is likely to be 7-bit ASCII. In that case, it is still just about as efficient to do brute-force search. The ciphertext is decrypted with the guessed key, and if all the 8th bits are zero (which will happen with an incorrect key with probability 1 in 256), then another block is decrypted. After several (say ten) blocks are decrypted, and the result always appears to be 7-bit ASCII, the key has a high probability of being correct. Current commercial DES chips do not lend themselves to doing exhaustive key search—they allow encrypting lots of data with a particular key. The relative speed of key loading is much less than the speed of encrypting data. However, it is straightforward to design and manufacture a key-searching DES chip. In 1977, Diffie and Hellman [DIFF77] did a detailed analysis of what it would cost to build a DES-breaking engine and concluded that for $20 million you could build a millionchip machine that could find a DES key in twelve hours (given a ⟨plaintext, ciphertext⟩ pair). In 1998, EFF (Electronic Frontier Foundation) [EFF98] built a special-purpose DES-breaking engine, called the EFF DES Cracker, for under $250K. It was designed to find a DES key in 4.5 days. With the design done, the cost of replicating the engine was under $150K. There are published papers [BIHA93] claiming that less straightforward attacks can break DES faster than simply searching the key space. However, these attacks involve the premise, unlikely in real-life situations, that the attacker can choose lots of plaintext and obtain the corresponding ciphertext. Still it is possible to encrypt multiple times with different keys (see §4.4 Multiple Encryption DES). It is generally believed that DES with triple encryption is 256 times as difficult to crack and therefore will be secure for the foreseeable future. 3.3.1 DATA ENCRYPTION STANDARD (DES) 59 The 64-bit input is subjected to an initial permutation to obtain a 64-bit result (which is just the input with the bits shuffled). The 56-bit key is used to generate sixteen 48-bit per-round keys, by taking a different 48-bit subset of the 56 bits for each of the keys. Each round takes as input the 64- bit output of the previous round, and the 48-bit per-round key, and produces a 64-bit output. After the 16th round, the 64-bit output has its halves swapped and is then subjected to another permutation, which happens to be the inverse of the initial permutation. That is the overview of how encryption works. Decryption works by essentially running DES backwards. To decrypt a block, you'd first run it through the initial permutation to undo the final permutation (the initial and final permutations are inverses of each other). You'd do the same key generation, though you'd use the keys in the opposite order (first use K16, the key you generated last). Then you run 16 rounds just like for encryption. Why this works will be explained when we explain what happens during a round. After 16 rounds of decryption, the output has its halves swapped and is then subjected to the final permutation (to undo the initial permutation). To fully specify DES, we need to specify the initial and final permutations, how the per round keys are generated, and what happens during a round. Let's start with the initial and final permutations of the data. 64-bit input 56-bit key Figure 3-2. Basic Structure of DES Initial Permutation Generate 16 per-round keys 48-bit K1 Round 1 48-bit K2 Round 2 · · · · · · · · · 48-bit K16 Round 16 swap left and right halves Final Permutation 64-bit output 60 SECRET KEY CRYPTOGRAPHY 3.3.2 3.3.2 The Permutations of the Data DES performs an initial and final permutation on the data, which do essentially nothing to enhance DES's security (see Why permute? on page 61). The most plausible reason for these permutations is to make DES less efficient to implement in software. The way the permutations are specified in the DES spec is as follows: The numbers in the above tables specify the bit numbers of the input to the permutation. The order of the numbers in the tables corresponds to the output bit position. So for example, the initial permutation moves input bit 58 to output bit 1 and input bit 50 to output bit 2. The permutation is not a random-looking permutation. Figure 3-3 pictures it. The arrows indicate the initial permutation. Reverse the arrows to get the final permutation. We hope you appreciate the time we spent staring at the numbers and discovering this completely useless structure. Initial Permutation (IP) Final Permutation (IP−1) 58 50 42 34 26 18 10 2 40 8 48 16 56 24 64 32 60 52 44 36 28 20 12 4 39 7 47 15 55 23 63 31 62 54 46 38 30 22 14 6 38 6 46 14 54 22 62 30 64 56 48 40 32 24 16 8 37 5 45 13 53 21 61 29 57 49 41 33 25 17 9 1 36 4 44 12 52 20 60 28 59 51 43 35 27 19 11 3 35 3 43 11 51 19 59 27 61 53 45 37 29 21 13 5 34 2 42 10 50 18 58 26 63 55 47 39 31 23 15 7 33 1 41 9 49 17 57 25 input bit output bit 12345678 12345678 octet 1 ABCDEFGH B 2 D 3 F 4 H 5 A 6 C 7 E 8 G Figure 3-3. Initial Permutation of Data Block 3.3.3 DATA ENCRYPTION STANDARD (DES) 61 The input is 8 octets. The output is 8 octets. The bits in the first octet of input get spread into the 8th bits of each of the octets. The bits in the second octet of input get spread into the 7th bits of all the octets. And in general, the bits of the i th octet get spread into the (9−i)th bits of all the octets. The pattern of spreading of the 8 bits in octet i of the input among the output octets is that the evennumbered bits go into octets 1–4, and the odd-numbered bits go into octets 5–8. Note that if the data happens to be 7-bit ASCII, with the top bit set to zero, then after the permutation the entire 5th octet will be zero. Since the permutation appears to have no security value, it seems nearly certain that there is no security significance to this particular permutation. 3.3.3 Generating the Per-Round Keys Next we'll specify how the sixteen 48-bit per-round keys are generated from the DES key. The DES key looks like it's 64 bits long, but 8 of the bits are parity. Let's number the bits of the DES key from left to right as 1, 2,...64. Bits 8, 16,...64 are the parity bits. DES performs a function, which we are about to specify, on these 64 bits to generate sixteen 48-bit keys, which are K1, K2,...K16. First it does an initial permutation on the 56 useful bits of the key, to generate a 56-bit output, which it divides into two 28-bit values, called C0 and D0. The permutation is specified as Why permute? Why can't the initial and final permutations of the data be of security value? Well, suppose they were important, i.e., if DES did not have them it would be possible to break DES. Let's call a modified DES that does not have the initial and final permutation EDS. Let's say we can break EDS, i.e., given a ⟨plaintext,ciphertext⟩ EDS pair, we can easily calculate the EDS key that converts the plaintext into the ciphertext. In that case, we can easily break DES as well. Given a DES ⟨plaintext,ciphertext⟩ pair ⟨m,c⟩, we simply do the inverse of the initial permutation (i.e. the final permutation) on m to get m', and the inverse of the final permutation (i.e. the initial permutation) on c to get c', and feed ⟨m',c'⟩ to our EDS-breaking code. The resulting EDS key will work as the DES key for ⟨m,c⟩. Note that when multiple encryptions of DES are being performed, the permutation might have some value. However, if encryption with key1 is followed by encryption with key2, then the final permutation following encryption with key1 will cancel the initial permutation for key2. That is one of the reasons people discuss alternating encrypt operations with decrypt operations (see §4.4 Multiple Encryption DES). In §3.3.3 Generating the Per-Round Keys, we'll see there is also a permutation of the key. It also has no security value (by a similar argument). 62 SECRET KEY CRYPTOGRAPHY 3.3.3 The way to read the table above is that the leftmost bit of the output is obtained by extracting bit 57 from the key. The next bit is bit 49 of the key, and so forth, with the final bit of D0 being bit 4 of the key. Notice that none of the parity bits (8, 16,...64) is used in C0 or D0. This permutation is not random. Figure 3-4 pictures it. Feel free to draw in any arrows or other graphic aids to make it clearer. The initial and final permutations of the bits in the key have no security value (just like the initial and final permutations of the data), so the permutations didn't have to be random—the identity permutation would have done nicely. Now the generation of the Ki proceeds in 16 rounds (see Figure 3-5). The number of bits shifted is different in the different rounds. In rounds 1, 2, 9, and 16, it is a single-bit rotate left (with C0 D0 57 49 41 33 25 17 9 63 55 47 39 31 23 15 1 58 50 42 34 26 18 7 62 54 46 38 30 22 10 2 59 51 43 35 27 14 6 61 53 45 37 29 19 11 3 60 52 44 36 21 13 5 28 20 12 4 1 2 3 4 5 6 7 ⇒ 57 49 41 33 25 17 9 9 10 11 12 13 14 15 1 58 50 42 34 26 18 17 18 19 20 21 22 23 10 2 59 51 43 35 27 25 26 27 28 29 30 31 19 11 3 60 52 44 36 33 34 35 36 37 38

39 63 55 47 39 31 23 15 41 42 43 44 45 46 47 7 62 54 46 38 30 22 49 50 51 52 53 54 55 14 6 61 53 45 37 29 57 58 59 60 61 62 63 21 13 5 28 20 12 4 Figure 3-4. Initial Permutation of Key loop back for next round Ci−1 Di−1 rotate left rotate left 1 4 Ci Di 2 9 3 2 Ki Figure 3-5. Round i for generating Ki 3.3.4 DATA ENCRYPTION STANDARD (DES) 63 the bit shifted off the left end carried around and shifted into the right end). In the other rounds, it is a two-bit rotate left. The permutations in this case are likely to be of some security value. The permutation of Ci that produces the left half of Ki is the following. Note that bits 9, 18, 22, and 25 are discarded. The permutation of the rotated Di−1 that produces the right half of Ki is as follows (where the bits of the rotated Di−1 are numbered 29, 30,...56, and bits 35, 38, 43, and 54 are discarded). Each of the halves of Ki is 24 bits, so Ki is 48 bits long. 3.3.4 A DES Round Now let's look at what a single round of DES does. Figure 3-6 shows both how encryption and decryption work. In encryption, the 64-bit input is divided into two 32-bit halves called Ln and Rn. The round generates as output 32-bit quantities Ln+1 and Rn+1. The concatenation of Ln+1 and Rn+1 is the 64- bit output of the round. Ln+1 is simply Rn. Rn+1 is obtained as follows. First Rn and Kn are input to what we call a mangler function, which outputs a 32-bit quantity. That quantity is ⨁'d with Ln to obtain the new Rn+1. The mangler takes as input 32 bits of the data plus 48 bits of the key to produce a 32-bit output. Given the above, suppose you want to run DES backward, i.e. to decrypt something. Suppose you know Ln+1 and Rn+1. How do you get Ln and Rn? Well, Rn is just Ln+1. Now you know Rn, Ln+1, Rn+1 and Kn. You also know that Rn+1 equals Ln ⨁ mangler(Rn, Kn). You can compute mangler(Rn, Kn), since you know Rn and Kn. Now ⨁ that with Rn+1. The result will be Ln. Note that the mangler is never run backwards. DES is elegantly designed to be reversible without constraining the mangler function to be reversible. This design is permutation to obtain the left half of Ki: 14 17 11 24 1 5 3 28 15 6 21 10 23 19 12 4 26 8 16 7 27 20 13 2 permutation to obtain the right half of Ki: 41 52 31 37 47 55 30 40 51 45 33 48 44 49 39 56 34 53 46 42 50 36 29 32 64 SECRET KEY CRYPTOGRAPHY 3.3.5 due to Feistel [FEIS73]. Theoretically the mangler could map all values to zero, and it would still be possible to run DES backwards, but having the mangler function map all functions to zero would make DES pretty unsecure (see Homework Problem 5). If you examine Figure 3-6 carefully, you will see that decryption is identical to encryption with the 32-bit halves swapped. In other words, feeding Rn+1|Ln+1 into round n produces Rn |Ln as output. 3.3.5 The Mangler Function The mangler function takes as input the 32-bit Rn, which we'll simply call R, and the 48-bit Kn, which we'll call K, and produces a 32-bit output which, when ⨁'d with Ln, produces Rn+1 (the next R). The mangler function first expands R from a 32-bit value to a 48-bit value. It does this by breaking R into eight 4-bit chunks and then expanding each of those chunks to 6 bits by taking the 64-bit input 32-bit Ln 32-bit Rn Mangler Function Kn ⨁ 32-bit Ln+1 32-bit Rn+1 64-bit output Encryption 64-bit output 32-bit Ln 32-bit Rn Mangler Function Kn ⨁ 32-bit Ln+1 32-bit Rn+1 64-bit input Decryption Figure 3-6. DES Round 3.3.5 DATA ENCRYPTION STANDARD (DES) 65 adjacent bits and concatenating them to the chunk. The leftmost and rightmost bits of R are considered adjacent. The 48-bit K is broken into eight 6-bit chunks. Chunk i of the expanded R is ⨁'d with chunk i of K to yield a 6-bit output. That 6-bit output is fed into an S-box, a substitution which produces a 4-bit output for each possible 6-bit input. Since there are 64 possible input values (6 bits) and only 16 possible output values (4 bits), the S-box clearly maps several input values to the same output value. As it turns out, there are exactly four input values that map to each possible output value. There's even more pattern to it than that. Each S-box could be thought of as four separate 4-bit to 4- Figure 3-7. Expansion of R to 48 bits chunk i of R chunk i of K ⨁ S-Box i Figure 3-8. Chunk Transformation 66 SECRET KEY CRYPTOGRAPHY 3.3.5 bit S-boxes, with the inner 4 bits of the 6-bit chunk serving as input, and the outer 2 bits selecting which of the four 4-bit S-boxes to use. The S-boxes are specified as follows: Input bits 1 and 6

Input bits 2 thru 5 ↓ 0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111 00 1110 0100 1101 0001 0010 1111 1011 1000 0011 1010 0110 1100 0101 1001 0000 0111 01 0000 1111 0111 0100 1110 0010 1101 0001 1010 0110 1100 1011 1001 0101 0011 1000 10 0100 0001 1110 1000 1101 0110 0010 1011 1111 1100 1001 0111 0011 1010 0101 0000 11 1111 1100 1000 0010 0100 1001 0001 0111 0101 1011 0011 1110 1010 0000 0110 1101 Figure 3-9. Table of 4-bit outputs of S-box 1 (bits 1 thru 4) Input bits 7 and 12 Input bits 8 thru 11 ↓ 0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111 00 1111 0001 1000 1110 0110 1011 0011 0100 1001 0111 0010 1101 1100 0000 0101 1010 01 0011 1101 0100 0111 1111 0010 1000 1110 1100 0000 0001 1010 0110 1001 1011 0101 10 0000 1110 0111 1011 1010 0100 1101 0001 0101 1000 1100 0110 1001 0011 0010 1111 11 1101 1000 1010 0001 0011 1111 0100 0010 1011 0110 0111 1100 0000 0101 1110 1001 Figure 3-10. Table of 4-bit outputs of S-box 2 (bits 5 thru 8) Input bits 13 and 18 Input bits 14 thru 17 ↓ 0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111 00 1010 0000 1001 1110 0110 0011 1111 0101 0001 1101 1100 0111 1011 0100 0010 1000 01 1101 0111 0000 1001 0011 0100 0110 1010 0010 1000 0101 1110 1100 1011 1111 0001 10 1101 0110 0100 1001 1000 1111 0011 0000 1011 0001 0010 1100 0101 1010 1110 0111 11 0001 1010 1101 0000 0110 1001 1000 0111 0100 1111 1110 0011 1011 0101 0010 1100 Figure 3-11. Table of 4-bit outputs of S-box 3 (bits 9 thru 12) Input bits 19 and 24 Input bits 20 thru 23 ↓ 0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111 00 0111 1101 1110 0011 0000 0110 1001 1010 0001 0010 1000 0101 1011 1100 0100 1111 01 1101 1000 1011 0101 0110 1111 0000 0011 0100 0111 0010 1100 0001 1010 1110 1001 10 1010 0110 1001 0000 1100 1011 0111 1101 1111 0001 0011 1110 0101 0010 1000 0100 11 0011 1111 0000 0110 1010 0001 1101 1000 1001 0100 0101 1011 1100 0111 0010 1110 Figure 3-12. Table of 4-bit outputs of S-box 4 (bits 13 thru 16) 3.3.5 DATA ENCRYPTION STANDARD (DES) 67 The 4-bit output of each of the eight S-boxes is combined into a 32-bit quantity whose bits are then permuted. A permutation at this point is of security value to DES in order to ensure that the bits of the output of an S-box on one round of DES affects the input of multiple S-boxes on the next round. Without the permutation, an input bit on the left would mostly affect the output bits on the left. Input bits 25 and 30 Input bits 26 thru 29 ↓ 0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111 00 0010 1100 0100 0001 0111 1010 1011 0110 1000 0101 0011 1111 1101 0000 1110 1001 01 1110 1011 0010 1100 0100 0111 1101 0001 0101 0000 1111 1010 0011 1001 1000 0110 10 0100 0010 0001 1011 1010 1101 0111 1000 1111 1001 1100 0101 0110 0011 0000 1110 11 1011 1000 1100 0111 0001 1110 0010 1101 0110 1111 0000 1001 1010 0100 0101 0011 Figure 3-13. Table of 4-bit outputs of S-box 5 (bits 17 thru 20) Input bits 31 and 36 Input bits 32 thru 35 ↓ 0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111 00 1100 0001 1010 1111 1001 0010 0110 1000 0000 1101 0011 0100 1110 0111 0101 1011 01 1010 1111 0100 0010 0111 1100 1001 0101 0110 0001 1101 1110 0000 1011 0011 1000 10 1001 1110 1111 0101 0010 1000 1100 0011 0111 0000 0100 1010 0001 1101 1011 0110 11 0100 0011 0010 1100 1001 0101 1111 1010 1011 1110 0001 0111 0110 0000 1000 1101 Figure 3-14. Table of 4-bit outputs of S-box 6 (bits 21 thru 24) Input bits 37 and 42 Input bits 38 thru 41 ↓ 0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111 00 0100 1011 0010 1110 1111 0000 1000 1101 0011 1100 1001 0111 0101 1010 0110 0001 01 1101 0000 1011 0111 0100 1001 0001 1010 1110 0011 0101 1100 0010 1111 1000 0110 10 0001 0100 1011 1101 1100 0011 0111 1110 1010 1111 0110 1000 0000 0101 1001 0010 11 0110 1011 1101 1000 0001 0100 1010 0111 1001 0101 0000 1111 1110 0010 0011 1100 Figure 3-15. Table of 4-bit outputs of S-box 7 (bits 25 thru 28) Input bits 43 and 48 Input bits 44 thru 47 ↓ 0000 0001 0010 0011 0100 0101 0110 0111 1000 1001 1010 1011 1100 1101 1110 1111 00 1101 0010 1000 0100 0110 1111 1011 0001 1010 1001 0011 1110 0101 0000 1100 0111 01 0001 1111 1101 1000 1010 0011 0111 0100

1100 0101 0110 1011 0000 1110 1001 0010 10 0111 1011 0100 0001 1001 1100 1110 0010 0000 0110 1010 1101 1111 0011 0101 1000 11 0010 0001 1110 0111 0100 1010 1000 1101 1111 1100 1001 0000 0011 0101 0110 1011 Figure 3-16. Table of 4-bit outputs of S-box 8 (bits 29 thru 32) 68 SECRET KEY CRYPTOGRAPHY 3.3.6 The actual permutation used is very random looking (we can't find any nice patterns to make the permutation easy to visualize—it's possible a non-random looking permutation would not be as secure). The way to read this is that the 1st bit of output of the permutation is the 16th input bit, the 2nd output bit is the 7th input bit,...the 32nd output bit is the 25th input bit. 3.3.6 Weak and Semi-Weak Keys We include this section mainly for completeness. There are sixteen DES keys that the security community warns people against using, because they have strange properties. But the probability of randomly generating one of these keys is only 16/256, which in our opinion is nothing to worry about. It's probably equally insecure to use a key with a value less than a thousand, since an attacker might be likely to start searching for keys from the bottom. Remember from §3.3.3 Generating the Per-Round Keys that the key is subjected to an initial permutation to generate two 28-bit quantities, C0 and D0. The sixteen suspect keys are ones for which C0 and D0 are one of the four values: all ones, all zeroes, alternating ones and zeroes, alternating zeroes and ones. Since there are four possible values for each half, there are sixteen possibilities in all. The four weak keys are the ones for which each of C0 and D0 are all ones or all zeroes. Weak keys are their own inverses.* The remaining twelve keys are the semi-weak keys. Each is the inverse of one of the others. 3.3.7 What's So Special About DES? DES is actually quite simple, as is IDEA (which we'll explain next). One gets the impression that anyone could design a secret key encryption algorithm. Just take the bits, shuffle them, shuffle them some more, and you have an algorithm. In fact, however, these things are very mysterious. For example, the S-boxes seem totally arbitrary. Did anyone put any thought into exactly what substitutions each S-box should perform? Well, Biham and Shamir [BIHA91] have shown that with an incredibly trivial change to DES consisting of swapping S-box 3 with S-box 7, DES is about an order of magnitude less secure in the face of a specific (admittedly not very likely) attack. *Two keys are inverses if encrypting with one is the same as decrypting with the other. 16 7 20 21 29 12 28 17 1 15 23 26 5 18 31 10 2 8 24 14 32 27 3 9 19 13 30 6 22 11 4 25 Figure 3-17. Permutation of the 32 bits from the S-boxes 3.4 INTERNATIONAL DATA ENCRYPTION ALGORITHM (IDEA) 69 It is unfortunate that the design process for DES was not more public. We don't know if the particular details were well-chosen for strength, whether someone flipped coins, for instance, to construct the S-boxes, or even whether the particular details were well-chosen to have some sort of weakness that could only be exploited by someone involved in the design process. The claim for why the design process was kept secret, and it is a plausible claim, is that the DES designers knew about many kinds of cryptanalytic attacks, and that they specifically designed DES to be strong against all the ones they knew about. If they publicized the design process, they'd have to divulge all the cryptanalytic attacks they knew about, which would then further educate potential bad guys, which might make some cryptographic standards that were designed without this knowledge vulnerable. In the hash algorithms designed by Ron Rivest (MD2, MD4, MD5), in order to eliminate the suspicion that they might be specifically chosen to have secret weaknesses, constants that should be reasonably random were chosen through some demonstrable manner, for instance by being the digits of an irrational number such as . 3.4 INTERNATIONAL DATA ENCRYPTION ALGORITHM (IDEA) IDEA (International Data Encryption Algorithm) was originally called IPES (Improved Proposed Encryption Standard). It was developed by Xuejia Lai and James L. Massey of ETH Zuria. IDEA was designed to be efficient to compute in software. It encrypts a 64-bit block of plaintext into a 64-bit block of ciphertext using a 128-bit key. It was published in 1991, so cryptanalysts have had time to find weaknesses. So far none has been found, at least by the good guys (the ones who would publish their results). IDEA is similar to DES in some ways. Both of them operate in rounds, and both have a complicated mangler function that does not have to be reversible in order for decryption to work. Instead, the mangler function is run in the same direction for encryption as decryption, in both IDEA and DES. In fact, both DES and IDEA have the property that encryption and decryption are identical except for key expansion. With DES, the same keys are used in the reverse order (see Homework Problem 11); with IDEA, the encryption and decryption keys are related in a more complex manner. 3.4.1 Primitive Operations Each primitive operation in IDEA maps two 16-bit quantities into a 16-bit quantity. (In contrast, each DES S-box maps a 6-bit quantity into a 4-bit quantity.) IDEA uses three operations, all easy to 2 70 SECRET KEY CRYPTOGRAPHY 3.4.1 compute in software, to create a mapping. Furthermore, the operations are all reversible, which is important in order to run IDEA backwards (i.e., to decrypt). The operations are bitwise exclusive or (⊕), a slightly modified add (+), and a slightly modified multiply (⊗). The reason add and multiply can't be used in the ordinary way is that the result has to be 16 bits, which won't always be the case when adding or multiplying two 16-bit quantities. Addition in IDEA is done by throwing away carries, which is equivalent to saying addition is mod 216. Multiplication in IDEA is done by first calculating the 32-bit result, and then taking the remainder when divided by 216+1 (which can be done in a clever efficient manner). Multiplication mod 216+1 is reversible, in the sense that every number x between 1 and 216 has an inverse y (i.e., a number in the range 1 to 216 such that multiplication by y will "undo" multiplication by x), because 216+1 happens to be prime. There is one subtlety, though. The number 0, which can be expressed in 16 bits, would not have an inverse. And the number 216, which is in the proper range for mod 216+1 arithmetic, cannot be expressed in 16 bits. So both problems are solved by treating 0 as an encoding for 216. How are these operations reversible? Of course the operations are not reversible if all that is known is the 16-bit output. For instance, if we have inputs A and B, and perform ⊕ to obtain C, we can't find A and B from C alone. However, when running IDEA backwards we will have C and B, and will use that to obtain A. ⊕ is easy. If you know B and C, then you can simply do B ⊕ C to get 64-bit input 128-bit key Figure 3-18. Basic Structure of IDEA Key Expansion K1 K2 K3 K4 Round 1 K5 K6 Round 2 · · · · · · · · · K49 K50 K51 K52 Round 17 64-bit output 3.4.2 INTERNATIONAL DATA ENCRYPTION ALGORITHM (IDEA) 71 A. B is its own inverse with ⊕. + is easy, too. You compute −B (mod 216). If you know C and −B, then you can find A by doing C + −B. With ⊗ you find B−1 (mod 216+1) using Euclid's algorithm— see §7.4 Euclid's Algorithm, and you perform C ⊗ B−1 to get A. The only part of IDEA that isn't necessarily reversible is the mangler function, and it is truly marvelous to note how IDEA's design manages not to require a reversible mangler function (see §3.4.3.2 Even Round). 3.4.2 Key Expansion The 128-bit key is expanded into 52 16-bit keys, K1, K2,...K52. The key expansion is done differently for encryption than for decryption. Once the 52 keys are generated, the encryption and decryption operations are the same. The 52 encryption keys are generated by writing out the 128-bit key and, starting from the left, chopping off 16 bits at a time. This generates eight 16-bit keys (see Figure 3-19). The next eight keys are generated by starting at bit 25, and wrapping around to the beginning when the end is reached (see Figure 3-20). The next eight keys are generated by offsetting 25 more bits, and so forth, until 52 keys are generated. The last offset starts at bit 23 and only needs 4 keys, so bits 1 thru 22 and bits 87 thru 128 get used in keys once less than bits 23 thru 86. We'll discuss how to generate the 52 decryption keys after we finish describing IDEA. 128-bit key K1 K2 K3 K4 K5 K6 K7 K8 Figure 3-19. Generation of keys 1 through 8 128-bit key 128-bit key Figure 3-20. Generation of keys 9 through 16 K9 K10 K11 K12 K13 K14 K15 K16 72 SECRET KEY CRYPTOGRAPHY 3.4.3 3.4.3 One Round Like DES, IDEA is performed in rounds. It has 17 rounds, where the odd-numbered rounds are different from the even-numbered rounds. (Note that in other descriptions of IDEA, it is described as having 8 rounds, where those rounds do the work of two of our rounds. Our explanation is functionally equivalent. It's just that we think it's clearer to explain it as having 17 rounds.) Each round takes the input, a 64-bit quantity, and treats it as four 16-bit quantities, which we'll call Xa, Xb, Xc, and Xd. Mathematical functions are performed on Xa, Xb, Xc, Xd to yield new versions of Xa, Xb, Xc, Xd. The odd rounds use four of the Ki, which we'll call Ka, Kb, Kc, and Kd. The even rounds use two Ki, which we'll call Ke and Kf. So round one uses K1, K2, K3, K4 (i.e., in round 1, Ka = K1, Kb = K2, Kc = K3, Kd = K4). Round 2 uses K5 and K6 (i.e., in round 2, Ke = K5 and Kf = K6). Round 3 uses K7, K8, K9, K10 (Ka = K7 etc.). Round 4 uses K11 and K12, and so forth. An odd round, therefore has as input Xa, Xb, Xc, Xd and keys Ka, Kb, Kc, Kd. An even round has as input Xa, Xb, Xc, Xd and keys Ke and Kf. 3.4.3.1 Odd Round The odd round is simple. Xa is replaced by Xa ⊗ Ka. Xd is replaced by Xd ⊗ Kd. Xc is replaced by Xb + Kb. Xb is replaced by Xc + Kc. Warning! If you're actually going to implement this, we lied a bit above because there's a strange quirk in IDEA. Possibly due to someone mixing up the labels on a diagram of IDEA, the keys K50 and K51 are swapped. That means that an implementation has to swap encryption keys K50 and K51 after generating them as described above. Xa Xb Xc Xd ⊗ Ka + Kb + Kc ⊗ Kd Xa Xb Xc Xd Figure 3-21. IDEA Odd Round 3.4.3.2 INTERNATIONAL DATA ENCRYPTION ALGORITHM (IDEA) 73 Note that this is easily reversible. To get from the new Xa to the old Xa, we perform ⊗ with the multiplicative inverse of Ka, mod 216+1. Likewise with Xd. To get the old Xb, given the new Xc, we add the additive inverse of Kb, i.e., we subtract Kb. So when decrypting, the odd rounds run as before, but with the mathematical inverses of the keys. This will undo the work that was done during that round in encryption. 3.4.3.2 Even Round The even round is a little more complicated. Again, we have Xa, Xb, Xc, and Xd. We have two keys, Ke and Kf. We're going to first compute two values, which we'll call Yin and Zin. We'll do a function, which we'll call the mangler function, which takes as input Yin, Zin, Ke, and Kf and produces what we'll call Yout and Zout. We'll use Yout and Zout to modify Xa, Xb, Xc, and Xd. Yin = Xa ⊕ Xb Zin = Xc ⊕ Xd Yout = ((Ke ⊗ Yin) + Zin) ⊗ Kf Zout = (Ke ⊗ Yin) + Yout Now we compute the new Xa, Xb, Xc, and Xd. new Xa = Xa ⊕ Yout new Xb = Xb ⊕ Yout new Xc = Xc ⊕ Zout new Xd = Xd ⊕ Zout Xa Xb Xc Xd Figure 3-22. IDEA Even Round ⊕ ⊕ Yin Zin Mangler Function Ke Kf Yout Zout ⊕ ⊕ Yout = ((Ke ⊗ Yin) + Zin) ⊗ Kf Zout = (Ke ⊗ Yin) + Yout ⊕ ⊕ Xa Xb Xc Xd 74 SECRET KEY CRYPTOGRAPHY 3.4.4 How is the work of an even round reversed? This is truly spectacular (we don't get out much). The even round is its own inverse! When performing decryption, the same keys are used as when performing encryption (not the mathematical inverses of the keys, as in the odd rounds). The even round takes as input the four quantities Xa, Xb, Xc, and Xd, together with keys Ke and Kf, and produces new Xa, new Xb, new Xc, and new Xd. If new Xa, new Xb, new Xc, and new Xd (with the same Ke and Kf) are fed into the even round, the output is the old Xa, Xb, Xc, and Xd. Why is this true? Note that new Xa = Xa ⊕ Yout and new Xb = Xb ⊕ Yout. In the beginning of the round, Xa and Xb are ⊕'d together, and the result is Yin, the input to the mangler function. What if we use new Xa and new Xb instead of Xa and Xb? (new Xa) ⊕ (new Xb) = (Xa ⊕ Yout) ⊕ (Xb ⊕ Yout) = Xa ⊕ Xb. So Yin will be the same, whether Xa and Xb are the inputs, or new Xa and new Xb are the inputs. The same is true for Xc and Xd. (Zin is the same value whether the inputs Xc and Xd are used, or inputs new Xc and new Xd are used.) So we've shown that the input to the mangler function is the same whether the input is Xa, Xb, Xc, and Xd or whether the input is new Xa, new Xb, new Xc, and new Xd. That means the output of the mangler function will be the same whether you're doing encryption (starting with Xa, etc.) or decryption (starting with new Xa, etc.). We called the outputs of the mangler function Yout and Zout. To get the first output of the round (new Xa, in the case of encryption), we take the first input (Xa) and ⊕ it with Yout. We're going to show that with inputs of new Xa, new Xb, new Xc, and new Xd, the output of the round is Xa, Xb, Xc, and Xd, i.e., that running the round with the output results in getting the input back. We'll use as inputs new Xa, new Xb, new Xc, and new Xd, and we know that Yout and Zout are the same as they would have been with inputs of Xa, Xb, Xc, and Xd. What happens in the round? The first output of the round is computed by taking the first input and ⊕ing it with Yout. We also know (from the encryption round) that new Xa = Xa ⊕ Yout. first output = first input ⊕ Yout first output = (new Xa) ⊕ Yout first output = (Xa ⊕ Yout) ⊕ Yout = Xa Magic! With an input of new Xa, we get an output of Xa. 3.4.4 Inverse Keys for Decryption IDEA is cleverly designed so that the same code (or hardware) can perform either encryption or decryption given different expanded keys. We want to compute inverse keys such that the encryption procedure, unmodified, will work as a decryption procedure. The basic idea is to take the inverses of the encryption keys and use them in the opposite order (use the inverse of the last-used encryption key as the first key used when doing decryption). 3.4.5 ADVANCED ENCRYPTION STANDARD (AES) 75 Remember that for encryption, we generated 52 keys, K1 through K52. We use four of them in each of the odd rounds, and two of them in each of the even rounds. And since we are working backwards, the first decryption keys should be inverses of the last-used encryption keys. Given that the final keys used are K49, K50, K51, and K52, in an odd round, the first four decryption keys will be inverses of the keys K49–K52. K49 is used in ⊗, so the decryption key K1 will be the multiplicative inverse of K49 mod 216+1. And the decryption key K4 is the multiplicative inverse of K52. Decryption keys K2 and K3 are the additive inverses of K50 and K51 (meaning negative K50 and K51). In the even rounds, as we explained, the keys do not have to be inverted. The same keys are used for encryption as decryption. 3.4.5 Does IDEA Work? The definition of "working" is that decryption really does undo encryption, and that is easy to see. Whether it is secure or not depends on the Fundamental Tenet of Cryptography, as nobody has yet published results on how to break it. Certainly, breaking IDEA by exhaustive search for the 128-bit key requires currently unbelievable computing resources. 3.5 ADVANCED ENCRYPTION STANDARD (AES) The world needed a new secret key standard. DES's key was too small. Triple DES (3DES, see §4.4 Multiple Encryption DES) was too slow. IDEA was encumbered (i.e., had patent protection), suspect, and slow. The National Institute of Standards and Technology (NIST) decided it wanted to facilitate creation of a new standard, but it had at least as difficult a political problem as a technical problem. Years of some branches of the U.S. government trying everything they could to hinder deployment of secure cryptography was likely to raise strong skepticism if a branch of the government stepped forward and said We're from the government, and we're here to help you develop and deploy strong crypto. The following quote gives a hint as to the deep resentment and mistrust that previous crypto export policies and the threat of domestic controls on encryption created. The NSA regularly lies to people who ask it for advice on export control. They have no reason not to; accomplishing their goal by any legal means is fine by them. Lying by government employees is legal. —John Gilmore 76 SECRET KEY CRYPTOGRAPHY 3.5.1 NIST really did want to help create an excellent new security standard. The new standard should be efficient, flexible, secure, and unencumbered (free to implement). But how could it help create one? Staying out of the picture wouldn't help, since nobody else seemed to have the technical reputation and energy to lead the effort. Proposing an NSA-designed cipher, designed in secret, wouldn't work since everyone would speculate that there were trapdoors. So on January 2, 1997, NIST announced a contest to select a new encryption standard to be used for protecting sensitive, non-classified, U.S. government information. Proposals would be accepted from anyone, anywhere in the world. The candidate ciphers had to meet a bunch of requirements, including having a documented design rationale (and not just Here's a bunch of transforms we do on the data). Then there were several years in which conferences were held for presentation of papers analyzing the candidates. There was a group of highly motivated cryptographers (the authors of submitted entries) looking for flaws in the proposals. And NIST also ran tests of the candidates for performance and other characteristics. After lots of investigation and discussion in the cryptographic community, NIST chose an algorithm called Rijndael, named after the two Belgian cryptographers who developed and submitted it—Dr. Joan Daemen of Proton World International and Dr. Vincent Rijmen, a postdoctoral researcher in the Electrical Engineering Department (ESAT) of Katholieke Universiteit Leuven [DAE99]. As of 26 November 2001, AES, a standardization of Rijndael, is a Federal Information Processing Standard [FIPS01]. Rijndael provides for a variety of block and key sizes. These two parameters can be chosen independently from 128, 160, 192, 224, and 256 bits. (in particular, key size and block size can be different). AES mandates a block size of 128 bits, and a choice of key size from 128, 192, and 256 bits*, with the resulting versions imaginatively called AES-128, AES-192, and AES-256, respectively. We'll describe Rijndael, mentioning the AES parameters explicitly from time to time. Rijndael is based on some beautiful mathematics, but we'll leave the discussion of the mathematics to §8.5 Mathematics of Rijndael. Rijndael is similar to DES and IDEA in that there is a series of rounds which mangles a plaintext block into a ciphertext block, and a key expansion algorithm that takes the key and massages it into a bunch of round keys. 3.5.1 Basic Structure Rijndael allows a certain amount of flexibility by use of two independent parameters, with a third parameter derived from the other two: *We1,2 can't imagine anyone using 192 bits. 256 is probably not necessary, since 128 bits is really long enough. 256 gives "bragging rights" to those who believe more must be better and are willing to take a performance hit to claim their product is more secure because of having a larger key than the competitors. Anyone willing to take a performance hit in order to advertise a larger key will use 256 bits rather than 192. 3.5.1 ADVANCED ENCRYPTION STANDARD (AES) 77 • The block size, Nb. This is the number of 32-bit words (4-octet columns) in an encryption block. AES has Nb = 4, because its 128-bit block size is four 32-bit words. 4Nb octet input 4Nk octet key Figure 3-23. Basic Structure of Rijndael a0 a1 ... Key Expansion a2 a3 K0 ⊕ ... Round 1 K1 ⊕ ... · · · · · · · · · ... Round NR KNr ⊕ ... 4NB octet output 78 SECRET KEY CRYPTOGRAPHY 3.5.2 • The key size, Nk. This is the number of 32-bit words (4-octet columns) in an encryption key. AES-128 has Nk = 4, AES-192 has Nk = 6, AES-256 has Nk = 8. Rijndael allows any Nk between 4 and 8 inclusive. • The number of rounds Nr . This parameter is a function of the other two parameters. The number of rounds needs to be larger for longer keys so that breaking the encryption is as difficult as a brute-force attack at that key size. The number of rounds needs to be larger for bigger block sizes (and key sizes) to allow sufficient mixing so that each bit of a plaintext block (or key) has a complex effect on each bit of the resulting ciphertext block. So Rijndael specifies that Nr = 6 + max(Nb, Nk). This means that AES-128 has ten rounds, AES-192 has twelve rounds, and AES-256 has fourteen rounds. Rijndael (Figure 3-23) keeps a rectangular array of octets as its state. The state has Nb 4-octet columns. Initially, the state is filled column by column from the 4Nb-octet input block. The state is transformed in Nr rounds into a final state, which is then read out column by column as the output block. Before round 1, between rounds, and after round Nr is an ⊕, into the state, of the next 4Nb octets from the expanded key, read out as columns. Rounds 1 through Nr−1 comprise an identical sequence of operations, while round Nr omits one of them. The Rijndael key is a 4Nk-octet block. The key expansion algorithm flows the key, column by column, into Nk 4-octet columns, then proceeds to create additional columns until it has (Nr+1)Nb columns, the exact amount of expanded key required. Key expansion uses the same kinds of primitive operations as the rounds do. Rows, columns, and round keys are numbered starting at 0. Round numbers start at 1. 3.5.2 Primitive Operations Rijndael is based on four primitive operations. • ⊕ • An octet-for-octet substitution, called the S-box (see Figure 3-24) • A rearrangement of octets comprising rotating a row or column by some number of cells • An operation called MixColumn, which replaces a 4-octet column with another 4-octet column. MixColumn can be implemented with a single table (Figure 3-26) containing 256 4- octet columns. Each of the four octets in an input column is used as an index to retrieve a column from the table; each column retrieved from the table is rotated vertically so that its top octet is in the same row as the input octet; and the four rotated columns are ⊕'d together to produce the output column. (See Figure 3-25.) We'll describe MixColumn mathematically in §24.5 Mathematics of Rijndael. 3.5.2 ADVANCED ENCRYPTION STANDARD (AES) 79 right (low-order) nibble 0123456789abcdef left (high-order) nibble 0 63 7c 77 7b f2 6b 6f c5 30 01 67 2b fe d7 ab 76 1 ca 82 c9 7d fa 59 47 f0 ad d4 a2 af 9c a4 72 c0 2 b7 fd 93 26 36 3f f7 cc 34 a5 e5 f1 71 d8 31 15 3 04 c7 23 c3 18 96 05 9a 07 12 80 e2 eb 27 b2 75 4 09 83 2c 1a 1b 6e 5a a0 52 3b d6 b3 29 e3 2f 84 5 53 d1 00 ed 20 fc b1 5b 6a cb be 39 4a 4c 58 cf 6 d0 ef aa fb 43 4d 33 85 45 f9 02 7f 50 3c 9f a8 7 51 a3 40 8f 92 9d 38 f5 bc b6 da 21 10 ff f3 d2 8 cd 0c 13 ec 5f 97 44 17 c4 a7 7e 3d 64 5d 19 73 9 60 81 4f dc 22 2a 90 88 46 ee b8 14 de 5e 0b db a e0 32 3a 0a 49 06 24 5c c2 d3 ac 62 91 95 e4 79 b e7 c8 37 6d 8d d5 4e a9 6c 56 f4 ea 65 7a ae 08 c ba 78 25 2e 1c a6 b4 c6 e8 dd 74 1f 4b bd 8b 8a d 70 3e b5 66 48 03 f6 0e 61 35 57 b9 86 c1 1d 9e e e1 f8 98 11 69 d9 8e 94 9b 1e 87 e9 ce 55 28 df f 8c a1 89 0d bf e6 42 68 41 99 2d 0f b0 54 bb 16 Figure 3-24. Rijndael S-box 2b 56 ⊕ 42 d4 b3 ⊕ 2b ⊕ 4c de a7 ⊕ d4 ⊕ 2b ⊕ b4 ad 41 ⊕ de ⊕ d4 ⊕ 7d 36 lookups ad ⊕ de ⊕ 67 ad ⊕ 79 ec Figure 3-25. MixColumn using table-lookup (see Figure 3-26) 80 SECRET KEY CRYPTOGRAPHY 3.5.2 right (low-order) nibble 0123456789abcdef left (high-order) nibble 0 00 00 00 00 02 01 01 03 04 02 02 06 06 03 03 05 08 04 04 0c 0a 05 05 0f 0c 06 06 0a 0e 07 07 09 10 08 08 18 12 09 09 1b 14 0a 0a 1e 16 0b 0b 1d 18 0c 0c 14 1a 0d 0d 17 1c 0e 0e 12 1e 0f 0f 11 1 20 10 10 30 22 11 11 33 24 12 12 36 26 13 13 35 28 14 14 3c 2a 15 15 3f 2c 16 16 3a 2e 17 17 39 30 18 18 28 32 19 19 2b 34 1a 1a 2e 36 1b 1b 2d 38 1c 1c 24 3a 1d 1d 27 3c 1e 1e 22 3e 1f 1f 21 2 40 20 20 60 42 21 21 63 44 22 22 66 46 23 23 65 48 24 24 6c 4a 25 25 6f 4c 26 26 6a 4e 27 27 69 50 28 28 78 52 29 29 7b 54 2a 2a 7e 56 2b 2b 7d 58 2c 2c 74 5a 2d 2d 77 5c 2e 2e 72 5e 2f 2f 71 3 60 30 30 50 62 31 31 53 64 32 32 56 66 33 33 55 68 34 34 5c 6a 35 35 5f 6c 36 36 5a 6e 37 37 59 70 38 38 48 72 39 39 4b 74 3a 3a 4e 76 3b 3b 4d 78 3c 3c 44 7a 3d 3d 47 7c 3e 3e 42 7e 3f 3f 41 4 80 40 40 c0 82 41 41 c3 84 42 42 c6 86 43 43 c5 88 44 44 cc 8a 45 45 cf 8c 46 46 ca 8e 47 47 c9 90 48 48 d8 92 49 49 db 94 4a 4a de 96 4b 4b dd 98 4c 4c d4 9a 4d 4d d7 9c 4e 4e d2 9e 4f 4f d1 5 a0 50 50 f0 a2 51 51 f3 a4 52 52 f6 a6 53 53 f5 a8 54 54 fc aa 55 55 ff ac 56 56 fa ae 57 57 f9 b0 58 58 e8 b2 59 59 eb b4 5a 5a ee b6 5b 5b ed b8 5c 5c e4 ba 5d 5d e7 bc 5e 5e e2 be 5f 5f e1 6 c0 60 60 a0 c2 61 61 a3 c4 62 62 a6 c6 63 63 a5 c8 64 64 ac ca 65 65 af cc 66 66 aa ce 67 67 a9 d0 68 68 b8 d2 69 69 bb d4 6a 6a be d6 6b 6b bd d8 6c 6c b4 da 6d 6d b7 dc 6e 6e b2 de 6f 6f b1 7 e0 70 70 90 e2 71 71 93 e4 72 72 96 e6 73 73 95 e8 74 74 9c ea 75 75 9f ec 76 76 9a ee 77 77 99 f0 78 78 88 f2 79 79 8b f4 7a 7a 8e f6 7b 7b 8d f8 7c 7c 84 fa 7d 7d 87 fc 7e 7e 82 fe 7f 7f 81 8 1b 80 80 9b 19 81 81 98 1f 82 82 9d 1d 83 83 9e 13 84 84 97 11 85 85 94 17 86 86 91 15 87 87 92 0b 88 88 83 09 89 89 80 0f 8a 8a 85 0d 8b 8b 86 03 8c 8c 8f 01 8d 8d 8c 07 8e 8e 89 05 8f 8f 8a 9 3b 90 90 ab 39 91 91 a8 3f 92 92 ad 3d 93 93 ae 33 94 94 a7 31 95 95 a4 37 96 96 a1 35 97 97 a2 2b 98 98 b3 29 99 99 b0 2f 9a 9a b5 2d 9b 9b b6 23 9c 9c bf 21 9d 9d bc 27 9e 9e b9 25 9f 9f ba a 5b a0 a0 fb 59 a1 a1 f8 5f a2 a2 fd 5d a3 a3 fe 53 a4 a4 f7 51 a5 a5 f4 57 a6 a6 f1 55 a7 a7 f2 4b a8 a8 e3 49 a9 a9 e0 4f aa aa e5 4d ab ab e6 43 ac ac ef 41 ad ad ec 47 ae ae e9 45 af af ea b 7b b0 b0 cb 79 b1 b1 c8 7f b2 b2 cd 7d b3 b3 ce 73 b4 b4 c7 71 b5 b5 c4 77 b6 b6 c1 75 b7 b7 c2 6b b8 b8 d3 69 b9 b9 d0 6f ba ba d5 6d bb bb d6 63 bc bc df 61 bd bd dc 67 be be d9 65 bf bf da c 9b c0 c0 5b 99 c1 c1 58 9f c2 c2 5d 9d c3 c3 5e 93 c4 c4 57 91 c5 c5 54 97 c6 c6 51 95 c7 c7 52 8b c8 c8 43 89 c9 c9 40 8f ca ca 45 8d cb cb 46 83 cc cc 4f 81 cd cd 4c 87 ce ce 49 85 cf cf 4a d bb d0 d0 6b b9 d1 d1 68 bf d2 d2 6d bd d3 d3 6e b3 d4 d4 67 b1 d5 d5 64 b7 d6 d6 61 b5 d7 d7 62 ab d8 d8 73 a9 d9 d9 70 af da da 75 ad db db 76 a3 dc dc 7f a1 dd dd 7c a7 de de 79 a5 df df 7a e db e0 e0 3b d9 e1 e1 38 df e2 e2 3d dd e3 e3 3e d3 e4 e4 37 d1 e5 e5 34 d7 e6 e6 31 d5 e7 e7 32 cb e8 e8 23 c9 e9 e9 20 cf ea ea 25 cd eb eb 26 c3 ec ec 2f c1 ed ed 2c c7 ee ee 29 c5 ef ef 2a f fb f0 f0 0b f9 f1 f1 08 ff f2 f2 0d fd f3 f3 0e f3 f4 f4 07 f1 f5 f5 04 f7 f6 f6 01 f5 f7 f7 02 eb f8 f8 13 e9 f9 f9 10 ef fa fa 15 ed fb fb 16 e3 fc fc 1f e1 fd fd 1c e7 fe fe 19 e5 ff ff 1a Figure 3-26. MixColumn table 3.5.2.1 ADVANCED ENCRYPTION STANDARD (AES) 81 3.5.2.1 What about the inverse cipher? • ⊕ is its own inverse • The inverse S-box is just given by a different table (see Figure 3-27) • The inverse of rotating a row or column is just rotating it the same amount in the opposite direction • The inverse of MixColumn, called InvMixColumn, is just like MixColumn, but with a different table (Figure 3-28) So the inverse cipher can clearly be implemented by applying the inverses of the primitive operations comprising the cipher in the opposite sequence from that in the cipher. But as it turns out, due to various mathematical properties (and by clever design), the inverse cipher can be made to look just like the forward cipher but with inverse operations, and with the round keys not just in reverse order, but having InvMixColumn applied to all but the first and last of them. This will be explained in §24.5 Mathematics of Rijndael. right (low-order) nibble 0123456789abcdef left (high-order) nibble 0 52 09 6a d5 30 36 a5 38 bf 40 a3 9e 81 f3 d7 fb 1 7c e3 39 82 9b 2f ff 87 34 8e 43 44 c4 de e9 cb 2 54 7b 94 32 a6 c2 23 3d ee 4c 95 0b 42 fa c3 4e 3 08 2e a1 66 28 d9 24 b2 76 5b a2 49 6d 8b d1 25 4 72 f8 f6 64 86 68 98 16 d4 a4 5c cc 5d 65 b6 92 5 6c 70 48 50 fd ed b9 da 5e 15 46 57 a7 8d 9d 84 6 90 d8 ab 00 8c bc d3 0a f7 e4 58 05 b8 b3 45 06 7 d0 2c 1e 8f ca 3f 0f 02 c1 af bd 03 01 13 8a 6b 8 3a 91 11 41 4f 67 dc ea 97 f2 cf ce f0 b4 e6 73 9 96 ac 74 22 e7 ad 35 85 e2 f9 37 e8 1c 75 df 6e a 47 f1 1a 71 1d 29 c5 89 6f b7 62 0e aa 18 be 1b b fc 56 3e 4b c6 d2 79 20 9a db c0 fe 78 cd 5a f4 c 1f dd a8 33 88 07 c7 31 b1 12 10 59 27 80 ec 5f d 60 51 7f a9 19 b5 4a 0d 2d e5 7a 9f 93 c9 9c ef e a0 e0 3b 4d ae 2a f5 b0 c8 eb bb 3c 83 53 99 61 f 17 2b 04 7e ba 77 d6 26 e1 69 14 63 55 21 0c 7d Figure 3-27. Rijndael inverse S-box 82 SECRET KEY CRYPTOGRAPHY 3.5.2.1 right (low-order) nibble 0123456789abcdef left (high-order) nibble 0 00 00 00 00 0e 09 0d 0b 1c 12 1a 16 12 1b 17 1d 38 24 34 2c 36 2d 39 27 24 36 2e 3a 2a 3f 23 31 70 48 68 58 7e 41 65 53 6c 5a 72 4e 62 53 7f 45 48 6c 5c 74 46 65 51 7f 54 7e 46 62 5a 77 4b 69 1 e0 90 d0 b0 ee 99 dd bb fc 82 ca a6 f2 8b c7 ad d8 b4 e4 9c d6 bd e9 97 c4 a6 fe 8a ca af f3 81 90 d8 b8 e8 9e d1 b5 e3 8c ca a2 fe 82 c3 af f5 a8 fc 8c c4 a6 f5 81 cf b4 ee 96 d2 ba e7 9b d9 2 db 3b bb 7b d5 32 b6 70 c7 29 a1 6d c9 20 ac 66 e3 1f 8f 57 ed 16 82 5c ff 0d 95 41 f1 04 98 4a ab 73 d3 23 a5 7a de 28 b7 61 c9 35 b9 68 c4 3e 93 57 e7 0f 9d 5e ea 04 8f 45 fd 19 81 4c f0 12 3 3b ab 6b cb 35 a2 66 c0 27 b9 71 dd 29 b0 7c d6 03 8f 5f e7 0d 86 52 ec 1f 9d 45 f1 11 94 48 fa 4b e3 03 93 45 ea 0e 98 57 f1 19 85 59 f8 14 8e 73 c7 37 bf 7d ce 3a b4 6f d5 2d a9 61 dc 20 a2 4 ad 76 6d f6 a3 7f 60 fd b1 64 77 e0 bf 6d 7a eb 95 52 59 da 9b 5b 54 d1 89 40 43 cc 87 49 4e c7 dd 3e 05 ae d3 37 08 a5 c1 2c 1f b8 cf 25 12 b3 e5 1a 31 82 eb 13 3c 89 f9 08 2b 94 f7 01 26 9f 5 4d e6 bd 46 43 ef b0 4d 51 f4 a7 50 5f fd aa 5b 75 c2 89 6a 7b cb 84 61 69 d0 93 7c 67 d9 9e 77 3d ae d5 1e 33 a7 d8 15 21 bc cf 08 2f b5 c2 03 05 8a e1 32 0b 83 ec 39 19 98 fb 24 17 91 f6 2f 6 76 4d d6 8d 78 44 db 86 6a 5f cc 9b 64 56 c1 90 4e 69 e2 a1 40 60 ef aa 52 7b f8 b7 5c 72 f5 bc 06 05 be d5 08 0c b3 de 1a 17 a4 c3 14 1e a9 c8 3e 21 8a f9 30 28 87 f2 22 33 90 ef 2c 3a 9d e4 7 96 dd 06 3d 98 d4 0b 36 8a cf 1c 2b 84 c6 11 20 ae f9 32 11 a0 f0 3f 1a b2 eb 28 07 bc e2 25 0c e6 95 6e 65 e8 9c 63 6e fa 87 74 73 f4 8e 79 78 de b1 5a 49 d0 b8 57 42 c2 a3 40 5f cc aa 4d 54 8 41 ec da f7 4f e5 d7 fc 5d fe c0 e1 53 f7 cd ea 79 c8 ee db 77 c1 e3 d0 65 da f4 cd 6b d3 f9 c6 31 a4 b2 af 3f ad bf a4 2d b6 a8 b9 23 bf a5 b2 09 80 86 83 07 89 8b 88 15 92 9c 95 1b 9b 91 9e 9 a1 7c 0a 47 af 75 07 4c bd 6e 10 51 b3 67 1d 5a 99 58 3e 6b 97 51 33 60 85 4a 24 7d 8b 43 29 76 d1 34 62 1f df 3d 6f 14 cd 26 78 09 c3 2f 75 02 e9 10 56 33 e7 19 5b 38 f5 02 4c 25 fb 0b 41 2e a 9a d7 61 8c 94 de 6c 87 86 c5 7b 9a 88 cc 76 91 a2 f3 55 a0 ac fa 58 ab be e1 4f b6 b0 e8 42 bd ea 9f 09 d4 e4 96 04 df f6 8d 13 c2 f8 84 1e c9 d2 bb 3d f8 dc b2 30 f3 ce a9 27 ee c0 a0 2a e5 b 7a 47 b1 3c 74 4e bc 37 66 55 ab 2a 68 5c a6 21 42 63 85 10 4c 6a 88 1b 5e 71 9f 06 50 78 92 0d 0a 0f d9 64 04 06 d4 6f 16

1d c3 72 18 14 ce 79 32 2b ed 48 3c 22 e0 43 2e 39 f7 5e 20 30 fa 55 c ec 9a b7 01 e2 93 ba 0a f0 88 ad 17 fe 81 a0 1c d4 be 83 2d da b7 8e 26 c8 ac 99 3b c6 a5 94 30 9c d2 df 59 92 db d2 52 80 c0 c5 4f 8e c9 c8 44 a4 f6 eb 75 aa ff e6 7e b8 e4 f1 63 b6 ed fc 68 d 0c 0a 67 b1 02 03 6a ba 10 18 7d a7 1e 11 70 ac 34 2e 53 9d 3a 27 5e 96 28 3c 49 8b 26 35 44 80 7c 42 0f e9 72 4b 02 e2 60 50 15 ff 6e 59 18 f4 44 66 3b c5 4a 6f 36 ce 58 74 21 d3 56 7d 2c d8 e 37 a1 0c 7a 39 a8 01 71 2b b3 16 6c 25 ba 1b 67 0f 85 38 56 01 8c 35 5d 13 97 22 40 1d 9e 2f 4b 47 e9 64 22 49 e0 69 29 5b fb 7e 34 55 f2 73 3f 7f cd 50 0e 71 c4 5d 05 63 df 4a 18 6d d6 47 13 f d7 31 dc ca d9 38 d1 c1 cb 23 c6 dc c5 2a cb d7 ef 15 e8 e6 e1 1c e5 ed f3 07 f2 f0 fd 0e ff fb a7 79 b4 92 a9 70 b9 99 bb 6b ae 84 b5 62 a3 8f 9f 5d 80 be 91 54 8d b5 83 4f 9a a8 8d 46 97 a3 Figure 3-28. InvMixColumn table 3.5.3 ADVANCED ENCRYPTION STANDARD (AES) 83 3.5.3 Key Expansion Key expansion starts with the key arranged as Nk 4-octet columns (see Figure 3-29) and iteratively generates the next Nk columns of the expanded key (see Figure 3-30). To generate the ith set of Nk a Key k0 k1 k2 k3 k4 k5 k6 k7 ... k0 k4 set 0 k1 k5 ... k2 k6 k3 k7 Figure 3-29. Rijndael key expansion, creation of set 0 a Ci S ⊕ set i−1 ... S S S ⊕ set i ⊕ ... ⊕ Figure 3-30. Rijndael key expansion, iteration step, Nk ≤ 6 84 SECRET KEY CRYPTOGRAPHY 3.5.4 columns (i starts at 1; the 0th set is the supplied key), all that is needed is the (i−1)th set. Column 0 of the new set is gotten by rotating the last column of the (i−1)th set upward one cell, applying the S-box to each octet, and then ⊕ing a constant based on i (see Figure 3-31) into octet 0. The rest of the columns in the set are generated in turn by ⊕ing the previous column with the corresponding column from the previous [(i−1)th] set. There is one exception to this—if Nk > 6, then an additional step is required to finish generating column 4, namely the application of the S-box to each octet (see Figure 3-32). Key expansion terminates as soon as (Nr+1)Nb columns of expanded key have been generated; this may happen in the middle of a set. 3.5.4 Rounds Each round is an identical sequence of three operations: 1. Each octet of the state has the S-Box applied to it. 2. Row 1 of the state is rotated left 1 column. Row 2 of the state is rotated left 2+Nb/8 columns (2 if Nb < 8, 3 otherwise). Row 3 of the state is rotated left 3+Nb/7 columns (3 if Nb < 7, 4 otherwise). (Note that for AES, this simplifies to rotating row i left i columns.) 3. Each column of the state has MixColumn applied to it. Round Nr omits this operation. i = 1 thru 10: 1 2 4 8 10 20 40 80 1b 36 i = 11 thru 20: 6c d8 ab 4d 9a 2f 5e bc 63 c6 i = 21 thru 30: 97 35 6a d4 b3 7d fa ef c5 (91) Figure 3-31. Rijndael key-expansion constants Ci a S ⊕⊕⊕⊕ S ⊕⊕⊕ S S Figure 3-32. Difference in Rijndael key expansion step, Nk = 8 [and similarly for Nk = 7] 3.5.5 ADVANCED ENCRYPTION STANDARD (AES) 85 3.5.5 Inverse Rounds Since each operation is invertible, decryption can be done by performing the inverse of each operation in the opposite order from that for encryption, and using the round keys in the reverse order. But as we mentioned earlier, we can make decryption have the same structure as encryption. To do this, we not only have to use the round keys in the opposite order, but we have to apply InvMixColumn to each column of all but the initial and final round keys. Then each inverse round is an identical sequence of three operations: 1. Each octet of the state has the inverse S-Box applied to it. 2. Row 1 of the state is rotated right 1 column. Row 2 of the state is rotated right 2+Nb/8 columns (2 if Nb < 8, 3 otherwise). Row 3 of the state is rotated right 3+Nb/7 columns (3 if Nb < 7, 4 otherwise). (Note that for AES, this simplifies to rotating row i right i columns.) 3. Each column of the state has InvMixColumn applied to it. Round Nr omits this operation. 3.5.6 Optimization In the straightforward implementation of a round, each octet of the state undergoes a table-lookup to apply the S-Box, and the resulting octet is later used for another table-lookup as part of MixColumn. We can combine these into a single table-lookup that transforms an octet into the column found by applying the S-Box and looking up the result in the MixColumn table. We can further optimize a round by retaining the old state while we compute the new state. We initialize the new state to the round key. For each octet in the old state, we perform the combined table-lookup and ⊕ the resulting column (rotated downward by the old octet's row number) into the appropriate column of the new state. For the first row (row 0), this is the old octet's column. For the second row (row 1), it's the column one to the left.* For the third row (row 2), it's either two or three columns to the left (depending on whether or not Nb < 8); and for the last row (row 3), it's either three or four columns to the left (depending on whether or not Nb < 7). When we've processed all the old octets, we've completed the round (including the following ⊕). The last round is a little different. We initialize the new state to the round key. For each octet in the old state, we apply the S-Box by table-lookup and ⊕ the result into the appropriate octet of the new state. The destination octet is in the same row as the source octet, and its column depends on the source row and column in the same manner as for the other rounds. The inverse rounds can be optimized in the same manner (with left replaced by right). *Think of the columns as residing on an upright cylinder—the rightmost column is immediately to the left of the leftmost. 86 SECRET KEY CRYPTOGRAPHY 3.6 3.6 RC4 A long random (or pseudo-random) string used to encrypt a message with a simple ⊕ operation is known as a one-time pad. A stream cipher generates a one-time pad and applies it to a stream of plaintext with ⊕. RC4 is a stream cipher designed by Ron Rivest. RC4 was a trade secret, but was "outed" in 1994. As a result, it has been extensively analyzed and is considered secure as long as you discard the first few (say 256) octets of the generated pad. The algorithm is an extremely simple (and fast) generator of pseudo-random streams of octets. The key can be from 1 to 256 octets. Even with a minimal key (a single null octet), the generated pseudo-random stream passes all the usual randomness tests (and so makes a fine pseudorandom number generator—I3've used it for that purpose on numerous occasions). RC4 keeps 258 octets of state information, 256 octets of which are a permutation of 0, 1,...255 that is initially computed from the key and then altered as each pad octet is generated. The box on page 87 gives a complete C implementation of RC4 one-time pad generation. 3.7 HOMEWORK 1. Come up with as efficient an encoding as you can to specify a completely general one-to-one mapping between 64-bit input values and 64-bit output values. 2. Token cards display a number that changes periodically, perhaps every minute. Each such device has a unique secret key. A human can prove possession of a particular such device by entering the displayed number into a computer system. The computer system knows the secret keys of each authorized device. How would you design such a device? 3. How many DES keys, on the average, encrypt a particular plaintext block to a particular ciphertext block? 4. Make an argument as to why the initial permutation of the bits of the DES key cannot have any security value. 5. Suppose the DES mangler function mapped every 32-bit value to zero, regardless of the value of its input. What function would DES then compute? 6. Are all the 56 bits of the DES key used an equal number of times in the Ki? Specify, for each of the Ki, which bits are not used. 3.7 HOMEWORK 87 7. What would change in the DES description if keys were input as 56-bit quantities rather than 64-bit quantities? 8. Why is a DES weak key (see §3.3.6 Weak and Semi-Weak Keys) its own inverse? Hint: DES encryption and decryption are the same once the per-round keys are generated. RC4 typedef unsigned char uns8; typedef unsigned short uns16; static uns8 state[256], x, y; /* 258 octets of state information */ void rc4init (key, length) /* initialize for encryption / decryption */ uns8 *key; uns16 length; { int i; uns8 t; uns8 j; uns8 k = 0; for (i = 256; i--; ) state[i] = i; for (i = 0, j = 0; i < 256; i++, j = (j + 1) % length) t = state[i], state[i] = state[k += key[j] + t], state[k] = t; x = 0; y = 0; } uns8 rc4step () /* return next pseudo-random octet */ { uns8 t; t = state[y += state[++x]], state[y] = state[x], state[x] = t; return (state[state[x] + state[y]]); } 88 SECRET KEY CRYPTOGRAPHY 3.7 9. Why is each DES semi-weak key the inverse of another semi-weak key? 10. Class project: design and implement an efficient DES breaking system. (Hint: Have your system take as input a ⟨plaintext, ciphertext⟩ pair and a starting key number, and searches from that starting key