Python Cryptography Toolkit

Version 2.7a1

The Python Cryptography Toolkit describes a package containing various cryptographic modules for the Python programming language. This documentation assumes you have some basic knowledge about the Python language, but not necessarily about cryptography.

Introduction

Design Goals

The Python cryptography toolkit is intended to provide a reliable and stable base for writing Python programs that require cryptographic functions.

A central goal has been to provide a simple, consistent interface for similar classes of algorithms. For example, all block cipher objects have the same methods and return values, and support the same feedback modes. Hash functions have a different interface, but it too is consistent over all the hash functions available. Some of these interfaces have been codified as Python Enhancement Proposal documents, as PEP 247, "API for Cryptographic Hash Functions", and PEP 272, "API for Block Encryption Algorithms".

This is intended to make it easy to replace old algorithms with newer, more secure ones. If you're given a bit of portably-written Python code that uses the DES encryption algorithm, you should be able to use AES instead by simply changing `from Crypto.Cipher import DES` to `from Crypto.Cipher import AES`, and changing all references to `DES.new()` to `AES.new()`. It's also fairly simple to write your own modules that mimic this interface, thus letting you use combinations or permutations of algorithms.

Some modules are implemented in C for performance; others are written in Python for ease of modification. Generally, low-level functions like ciphers and hash functions are written in C, while less speed-critical functions have been written in Python. This division may change in future releases. When speeds are quoted in this document, they were measured on a 500 MHz Pentium II running Linux. The exact speeds will obviously vary with different machines,

different compilers, and the phase of the moon, but they provide a crude basis for comparison. Currently the cryptographic implementations are acceptably fast, but not spectacularly good. I welcome any suggestions or patches for faster code.

I have placed the code under no restrictions; you can redistribute the code freely or commercially, in its original form or with any modifications you make, subject to whatever local laws may apply in your jurisdiction. Note that you still have to come to some agreement with the holders of any patented algorithms you're using. If you're intensively using these modules, please tell me about it; there's little incentive for me to work on this package if I don't know of anyone using it.

I also make no guarantees as to the usefulness, correctness, or legality of these modules, nor does their inclusion constitute an endorsement of their effectiveness. Many cryptographic algorithms are patented; inclusion in this package does not necessarily mean you are allowed to incorporate them in a product and sell it. Some of these algorithms may have been cryptanalyzed, and may no longer be secure. While I will include commentary on the relative security of the algorithms in the sections entitled "Security Notes", there may be more recent analyses I'm not aware of. (Or maybe I'm just clueless.) If you're implementing an important system, don't just grab things out of a toolbox and put them together; do some research first. On the other hand, if you're just interested in keeping your co-workers or your relatives out of your files, any of the components here could be used.

This document is very much a work in progress. If you have any questions, comments, complaints, or suggestions, please send them to me.

Acknowledgements

Much of the code that actually implements the various cryptographic algorithms was not written by me. I'd like to thank all the people who implemented them, and released their work under terms which allowed me to use their code. These individuals are credited in the relevant chapters of this documentation. Bruce Schneier's book Applied Cryptography was also very useful in writing this toolkit; I highly recommend it if you're interested in learning more about cryptography.

Good luck with your cryptography hacking!

Crypto.Hash: Hash Functions

Hash functions take arbitrary strings as input, and produce an output of fixed size that is dependent on the input; it should never be possible to derive the input data given only the hash function's output. One simple hash function consists of simply adding together all the bytes of the input, and taking the

result modulo 256. For a hash function to be cryptographically secure, it must be very difficult to find two messages with the same hash value, or to find a message with a given hash value. The simple additive hash function fails this criterion miserably and the hash functions described below meet this criterion (as far as we know). Examples of cryptographically secure hash functions include MD2, MD5, and SHA1.

Hash functions can be used simply as a checksum, or, in association with a public-key algorithm, can be used to implement digital signatures.

The hashing algorithms currently implemented are:

Hash function	Digest length	Security
MD2	128 bits	Insecure, do not use
MD4	128 bits	Insecure, do not use
MD5	128 bits	Insecure, do not use
RIPEMD160	160 bits	Secure.
SHA1	160 bits	SHA1 is shaky. Walk, do not run, away from SHA1.
SHA256	256 bits	Secure.

Resources: On SHA1 (in)security: http://www.schneier.com/blog/archives/2005/02/cryptanalysis_o.html SHA1 phase-out by 2010: http://csrc.nist.gov/groups/ST/toolkit/documents/shs/hash_standards_comments.pdf On MD5 insecurity: http://www.schneier.com/blog/archives/2008/12/forging_ssl_cer.html

Crypto.Hash.HMAC implements the RFC-2104 HMAC algorithm. The HMAC module is a copy of Python 2.2's module, and works on Python 2.1 as well. HMAC's security depends on the cryptographic strength of the key handed to it, and on the underlying hashing method used. HMAC-MD5 and HMAC-SHA1 are used in IPSEC and TLS.

All hashing modules with the exception of HMAC share the same interface. After importing a given hashing module, call the `new()` function to create a new hashing object. You can now feed arbitrary strings into the object with the `update()` method, and can ask for the hash value at any time by calling the `digest()` or `hexdigest()` methods. The `new()` function can also be passed an optional string parameter that will be immediately hashed into the object's state.

To create a HMAC object, call HMAC's `'new()` function with the key (as a string or bytes object) to be used, an optional message, and the hash function to use. HMAC defaults to using MD5. This is not a secure default, please use SHA256 or better instead in new implementations.

Hash function modules define one variable:

digest_size: An integer value; the size of the digest produced by the hashing objects. You could also obtain this value by creating a sample object, and taking the length of the digest string it returns, but using `digest_size` is faster.

The methods for hashing objects are always the following:

copy(): Return a separate copy of this hashing object. An `update` to this copy won't affect the original object.

digest(): Return the hash value of this hashing object, as a string containing 8-bit data. The object is not altered in any way by this function; you can continue updating the object after calling this function. Python 3.x: digest() returns a bytes object

hexdigest(): Return the hash value of this hashing object, as a string containing the digest data as hexadecimal digits. The resulting string will be twice as long as that returned by `digest()`. The object is not altered in any way by this function; you can continue updating the object after calling this function.

update(arg): Update this hashing object with the string `arg`. Python 3.x: The passed argument must be an object interpretable as a buffer of bytes

Here's an example, using the SHA-256 algorithm:

```
>>> from Crypto.Hash import SHA256
>>> m = SHA256.new()
>>> m.update('abc')
>>> m.digest()
'\xbax\x16\xbf\x8f\x01\xcf\xeaAA@\xde]\xae"#\xb0\x03a\xa3\x96\x17z\x9c\xb4\x10\xffa\xf2\x00\
>>> m.hexdigest()
'ba7816bf8f01cfea414140de5dae2223b00361a396177a9cb410ff61f20015ad'
```

Here's an example of using HMAC:

```
>>> from Crypto.Hash import HMAC, SHA256
>>> m = HMAC.new('Please do not use this key in your code, with sugar on top',
        '', SHA256)
>>> m.update('abc')
>>> m.digest()
'F\xaa\x83\t\x97<\x8c\x12\xff\xe8l\xca:\x1d\xb4\xfc7\xfa\x84tK-\xb0\x00v*\xc2\x90\x19\xaa\xf
>>> m.hexdigest()
'46aa8309973c8c12ffe86cca3a1db4fc37fa84744b2db000762ac29019aafa7a'
```

Security Notes

Hashing algorithms are broken by developing an algorithm to compute a string that produces a given hash value, or to find two messages that produce the

same hash value. Consider an example where Alice and Bob are using digital signatures to sign a contract. Alice computes the hash value of the text of the contract and signs the hash value with her private key. Bob could then compute a different contract that has the same hash value, and it would appear that Alice signed that bogus contract; she'd have no way to prove otherwise. Finding such a message by brute force takes `pow(2, b-1)` operations, where the hash function produces b-bit hashes.

If Bob can only find two messages with the same hash value but can't choose the resulting hash value, he can look for two messages with different meanings, such as "I will mow Bob's lawn for $10" and "I owe Bob $1,000,000", and ask Alice to sign the first, innocuous contract. This attack is easier for Bob, since finding two such messages by brute force will take `pow(2, b/2)` operations on average. However, Alice can protect herself by changing the protocol; she can simply append a random string to the contract before hashing and signing it; the random string can then be kept with the signature.

Some of the algorithms implemented here have been completely broken. The MD2, MD4 and MD5 hash functions are widely considered insecure hash functions, as it has been proven that meaningful hash collisions can be generated for them, in the case of MD4 and MD5 in mere seconds. MD2 is rather slow at 1250 K/sec. MD4 is faster at 44,500 K/sec. MD5 is a strengthened version of MD4 with four rounds; beginning in 2004, a series of attacks were discovered and it's now possible to create pairs of files that result in the same MD5 hash. The MD5 implementation is moderately well-optimized and thus faster on x86 processors, running at 35,500 K/sec. MD5 may even be faster than MD4, depending on the processor and compiler you use. MD5 is still supported for compatibility with existing protocols, but implementors should use SHA256 in new software because there are no known attacks against SHA256.

All the MD* algorithms produce 128-bit hashes. SHA1 produces a 160-bit hash. Because of recent theoretical attacks against SHA1, NIST recommended phasing out use of SHA1 by 2010. SHA256 produces a larger 256-bit hash, and there are no known attacks against it. It operates at 10,500 K/sec. RIPEMD has a 160-bit output, the same output size as SHA1, and operates at 17,600 K/sec.

Credits

The MD2 and MD4 implementations were written by A.M. Kuchling, and the MD5 code was implemented by Colin Plumb. The SHA1 code was originally written by Peter Gutmann. The RIPEMD160 code as of version 2.1.0 was written by Dwayne Litzenberger. The SHA256 code was written by Tom St. Denis and is part of the LibTomCrypt library (http://www.libtomcrypt.org/); it was adapted for the toolkit by Jeethu Rao and Taylor Boon.

Crypto.Cipher: Encryption Algorithms

Encryption algorithms transform their input data, or **plaintext**, in some way that is dependent on a variable **key**, producing **ciphertext**. This transformation can easily be reversed, if (and, hopefully, only if) one knows the key. The key can be varied by the user or application and chosen from some very large space of possible keys.

For a secure encryption algorithm, it should be very difficult to determine the original plaintext without knowing the key; usually, no clever attacks on the algorithm are known, so the only way of breaking the algorithm is to try all possible keys. Since the number of possible keys is usually of the order of 2 to the power of 56 or 128, this is not a serious threat, although 2 to the power of 56 is now considered insecure in the face of custom-built parallel computers and distributed key guessing efforts.

Block ciphers take multibyte inputs of a fixed size (frequently 8 or 16 bytes long) and encrypt them. Block ciphers can be operated in various modes. The simplest is Electronic Code Book (or ECB) mode. In this mode, each block of plaintext is simply encrypted to produce the ciphertext. This mode can be dangerous, because many files will contain patterns greater than the block size; for example, the comments in a C program may contain long strings of asterisks intended to form a box. All these identical blocks will encrypt to identical ciphertext; an adversary may be able to use this structure to obtain some information about the text.

To eliminate this weakness, there are various feedback modes in which the plaintext is combined with the previous ciphertext before encrypting; this eliminates any repetitive structure in the ciphertext.

One mode is Cipher Block Chaining (CBC mode); another is Cipher FeedBack (CFB mode). CBC mode still encrypts in blocks, and thus is only slightly slower than ECB mode. CFB mode encrypts on a byte-by-byte basis, and is much slower than either of the other two modes. The chaining feedback modes require an initialization value to start off the encryption; this is a string of the same length as the ciphering algorithm's block size, and is passed to the `new()` function.

The currently available block ciphers are listed in the following table, and are in the `Crypto.Cipher` package:

Cipher	Key Size/Block Size
AES	16, 24, or 32 bytes/16 bytes
ARC2	Variable/8 bytes
Blowfish	Variable/8 bytes
CAST	Variable/8 bytes

Cipher	Key Size/Block Size
DES	8 bytes/8 bytes
DES3 (Triple DES)	16 bytes/8 bytes

In a strict formal sense, **stream ciphers** encrypt data bit-by-bit; practically, stream ciphers work on a character-by-character basis. Stream ciphers use exactly the same interface as block ciphers, with a block length that will always be 1; this is how block and stream ciphers can be distinguished. The only feedback mode available for stream ciphers is ECB mode.

The currently available stream ciphers are listed in the following table:

Cipher	Key Size
ARC4	Variable
XOR	Variable

ARC4 is short for "Alleged RC4". In September of 1994, someone posted C code to both the Cypherpunks mailing list and to the Usenet newsgroup `sci.crypt`, claiming that it implemented the RC4 algorithm. This claim turned out to be correct. Note that there's a damaging class of weak RC4 keys; this module won't warn you about such keys.

A similar anonymous posting was made for Alleged RC2 in January, 1996.

An example usage of the DES module:

```
>>> from Crypto.Cipher import DES
>>> obj=DES.new('abcdefgh', DES.MODE_ECB)
>>> plain="Guido van Rossum is a space alien."
>>> len(plain)
34
>>> obj.encrypt(plain)
Traceback (innermost last):
  File "<stdin>", line 1, in ?
ValueError: Strings for DES must be a multiple of 8 in length
>>> ciph=obj.encrypt(plain+'XXXXXX')
>>> ciph
'\021,\343Nq\214DY\337T\342pA\372\255\311s\210\363,\300j\330\250\312\347\342I\3215w\03561\30
>>> obj.decrypt(ciph)
'Guido van Rossum is a space alien.XXXXXX'
```

All cipher algorithms share a common interface. After importing a given module, there is exactly one function and two variables available.

new(key, mode[, IV]): Returns a ciphering object, using `key` and feedback mode `mode`. If `mode` is `MODE_CBC` or `MODE_CFB`, `IV` must be provided, and must be a string of the same length as the block size. Some algorithms support additional keyword arguments to this function; see the "Algorithm-specific Notes for Encryption Algorithms" section below for the details. Python 3.x: `‘mode` is a string object; `‘key‘` and `‘IV‘` must be objects interpretable as a buffer of bytes.

block_size: An integer value; the size of the blocks encrypted by this module. Strings passed to the `encrypt` and `decrypt` functions must be a multiple of this length. For stream ciphers, `block_size` will be 1.

key_size: An integer value; the size of the keys required by this module. If `key_size` is zero, then the algorithm accepts arbitrary-length keys. You cannot pass a key of length 0 (that is, the null string `""` as such a variable-length key.

All cipher objects have at least three attributes:

block_size: An integer value equal to the size of the blocks encrypted by this object. Identical to the module variable of the same name.

IV: Contains the initial value which will be used to start a cipher feedback mode. After encrypting or decrypting a string, this value will reflect the modified feedback text; it will always be one block in length. It is read-only, and cannot be assigned a new value. Python 3.x: `‘IV‘` is a bytes object.

key_size: An integer value equal to the size of the keys used by this object. If `key_size` is zero, then the algorithm accepts arbitrary-length keys. For algorithms that support variable length keys, this will be 0. Identical to the module variable of the same name.

All ciphering objects have the following methods:

decrypt(string): Decrypts `string`, using the key-dependent data in the object, and with the appropriate feedback mode. The string's length must be an exact multiple of the algorithm's block size. Returns a string containing the plaintext. Python 3.x: decrypt() will return a bytes object.

Note: Do not use the same cipher object for both encryption an decryption, since both operations share the same IV buffer, so the results will probably not be what you expect.

encrypt(string): Encrypts a non-null `string`, using the key-dependent data in the object, and with the appropriate feedback mode. The string's length must be an exact multiple of the algorithm's block size; for stream ciphers, the string can be of any length. Returns a string containing the ciphertext. Python 3.x: `‘string‘` must be an object interpretable as a buffer of bytes. encrypt() will return a bytes object.

Note: Do not use the same cipher object for both encryption an decryption, since both operations share the same IV buffer, so the results will probably not be what you expect.

Security Notes

Encryption algorithms can be broken in several ways. If you have some ciphertext and know (or can guess) the corresponding plaintext, you can simply try every possible key in a **known-plaintext** attack. Or, it might be possible to encrypt text of your choice using an unknown key; for example, you might mail someone a message intending it to be encrypted and forwarded to someone else. This is a **chosen-plaintext** attack, which is particularly effective if it's possible to choose plaintexts that reveal something about the key when encrypted.

Stream ciphers are only secure if any given key is never used twice. If two (or more) messages are encrypted using the same key in a stream cipher, the cipher can be broken fairly easily.

DES (5100 K/sec) has a 56-bit key; this is starting to become too small for safety. It has been shown in 2009 that a ~$10,000 machine can break DES in under a day on average. NIST has withdrawn FIPS 46-3 in 2005. DES3 (1830 K/sec) uses three DES encryptions for greater security and a 112-bit or 168-bit key, but is correspondingly slower. Attacks against DES3 are not currently feasible, and it has been estimated to be useful until 2030. Bruce Schneier endorses DES3 for its security because of the decades of study applied against it. It is, however, slow.

There are no known attacks against Blowfish (9250 K/sec) or CAST (2960 K/sec), but they're all relatively new algorithms and there hasn't been time for much analysis to be performed; use them for serious applications only after careful research.

pycrypto implements CAST with up to 128 bits key length (CAST-128). This algorithm is considered obsoleted by CAST-256. CAST is patented by Entrust Technologies and free for non-commercial use.

Bruce Schneier recommends his newer Twofish algorithm over Blowfish where a fast, secure symmetric cipher is desired. Twofish was an AES candidate. It is slightly slower than Rijndael (the chosen algorithm for AES) for 128-bit keys, and slightly faster for 256-bit keys.

AES, the Advanced Encryption Standard, was chosen by the US National Institute of Standards and Technology from among 6 competitors, and is probably your best choice. It runs at 7060 K/sec, so it's among the faster algorithms around.

ARC4 ("Alleged" RC4) (8830 K/sec) has been weakened. Specifically, it has been shown that the first few bytes of the ARC4 keystream are strongly non-random,

leaking information about the key. When the long-term key and nonce are merely concatenated to form the ARC4 key, such as is done in WEP, this weakness can be used to discover the long-term key by observing a large number of messages encrypted with this key. Because of these possible related-key attacks, ARC4 should only be used with keys generated by a strong RNG, or from a source of sufficiently uncorrelated bits, such as the output of a hash function. A further possible defense is to discard the initial portion of the keystream. This altered algorithm is called RC4-drop(n). While ARC4 is in wide-spread use in several protocols, its use in new protocols or applications is discouraged.

ARC2 ("Alleged" RC2) is vulnerable to a related-key attack, 2^34 chosen plaintexts are needed. Because of these possible related-key attacks, ARC2 should only be used with keys generated by a strong RNG, or from a source of sufficiently uncorrelated bits, such as the output of a hash function.

Credits

The code for Blowfish was written from scratch by Dwayne Litzenberger, based on a specification by Bruce Schneier, who also invented the algorithm; the Blowfish algorithm has been placed in the public domain and can be used freely. (See http://www.schneier.com/paper-blowfish-fse.html for more information about Blowfish.) The CAST implementation was written by Wim Lewis. The DES implementation uses libtomcrypt, which was written by Tom St Denis.

The Alleged RC4 code was posted to the `sci.crypt` newsgroup by an unknown party, and re-implemented by A.M. Kuchling.

Crypto.Protocol: Various Protocols

Crypto.Protocol.AllOrNothing

This module implements all-or-nothing package transformations. An all-or-nothing package transformation is one in which some text is transformed into message blocks, such that all blocks must be obtained before the reverse transformation can be applied. Thus, if any blocks are corrupted or lost, the original message cannot be reproduced.

An all-or-nothing package transformation is not encryption, although a block cipher algorithm is used. The encryption key is randomly generated and is extractable from the message blocks.

AllOrNothing(ciphermodule, mode=None, IV=None): Class implementing the All-or-Nothing package transform.

`ciphermodule` is a module implementing the cipher algorithm to use. Optional arguments `mode` and `IV` are passed directly through to the `ciphermodule.new()`

method; they are the feedback mode and initialization vector to use. All three arguments must be the same for the object used to create the digest, and to undigest'ify the message blocks.

The module passed as `ciphermodule` must provide the PEP 272 interface. An encryption key is randomly generated automatically when needed.

The methods of the `AllOrNothing` class are:

digest(text): Perform the All-or-Nothing package transform on the string `text`. Output is a list of message blocks describing the transformed text, where each block is a string of bit length equal to the cipher module's block_size.

undigest(mblocks): Perform the reverse package transformation on a list of message blocks. Note that the cipher module used for both transformations must be the same. `mblocks` is a list of strings of bit length equal to `ciphermodule`'s block_size. The output is a string object.

Crypto.Protocol.Chaffing

Winnowing and chaffing is a technique for enhancing privacy without requiring strong encryption. In short, the technique takes a set of authenticated message blocks (the wheat) and adds a number of chaff blocks which have randomly chosen data and MAC fields. This means that to an adversary, the chaff blocks look as valid as the wheat blocks, and so the authentication would have to be performed on every block. By tailoring the number of chaff blocks added to the message, the sender can make breaking the message computationally infeasible. There are many other interesting properties of the winnow/chaff technique.

For example, say Alice is sending a message to Bob. She packetizes the message and performs an all-or-nothing transformation on the packets. Then she authenticates each packet with a message authentication code (MAC). The MAC is a hash of the data packet, and there is a secret key which she must share with Bob (key distribution is an exercise left to the reader). She then adds a serial number to each packet, and sends the packets to Bob.

Bob receives the packets, and using the shared secret authentication key, authenticates the MACs for each packet. Those packets that have bad MACs are simply discarded. The remainder are sorted by serial number, and passed through the reverse all-or-nothing transform. The transform means that an eavesdropper (say Eve) must acquire all the packets before any of the data can be read. If even one packet is missing, the data is useless.

There's one twist: by adding chaff packets, Alice and Bob can make Eve's job much harder, since Eve now has to break the shared secret key, or try every combination of wheat and chaff packet to read any of the message. The cool thing is that Bob doesn't need to add any additional code; the chaff packets are already filtered out because their MACs don't match (in all likelihood – since

the data and MACs for the chaff packets are randomly chosen it is possible, but very unlikely that a chaff MAC will match the chaff data). And Alice need not even be the party adding the chaff! She could be completely unaware that a third party, say Charles, is adding chaff packets to her messages as they are transmitted.

Chaff(factor=1.0, blocksper=1): Class implementing the chaff adding algorithm. `factor` is the number of message blocks to add chaff to, expressed as a percentage between 0.0 and 1.0; the default value is 1.0. `blocksper` is the number of chaff blocks to include for each block being chaffed, and defaults to 1. The default settings add one chaff block to every message block. By changing the defaults, you can adjust how computationally difficult it could be for an adversary to brute-force crack the message. The difficulty is expressed as:

```
pow(blocksper, int(factor * number-of-blocks))
```

For ease of implementation, when `factor` < 1.0, only the first `int(factor*number-of-blocks)` message blocks are chaffed.

`Chaff` instances have the following methods:

chaff(blocks): Add chaff to message blocks. `blocks` is a list of 3-tuples of the form `(serial-number, data, MAC)`.

Chaff is created by choosing a random number of the same byte-length as `data`, and another random number of the same byte-length as `MAC`. The message block's serial number is placed on the chaff block and all the packet's chaff blocks are randomly interspersed with the single wheat block. This method then returns a list of 3-tuples of the same form. Chaffed blocks will contain multiple instances of 3-tuples with the same serial number, but the only way to figure out which blocks are wheat and which are chaff is to perform the MAC hash and compare values.

Crypto.PublicKey: Public-Key Algorithms

So far, the encryption algorithms described have all been *private key* ciphers. The same key is used for both encryption and decryption so all correspondents must know it. This poses a problem: you may want encryption to communicate sensitive data over an insecure channel, but how can you tell your correspondent what the key is? You can't just e-mail it to her because the channel is insecure. One solution is to arrange the key via some other way: over the phone or by meeting in person.

Another solution is to use **public-key** cryptography. In a public key system, there are two different keys: one for encryption and one for decryption. The encryption key can be made public by listing it in a directory or mailing it to your

correspondent, while you keep the decryption key secret. Your correspondent then sends you data encrypted with your public key, and you use the private key to decrypt it. While the two keys are related, it's very difficult to derive the private key given only the public key; however, deriving the private key is always possible given enough time and computing power. This makes it very important to pick keys of the right size: large enough to be secure, but small enough to be applied fairly quickly.

Many public-key algorithms can also be used to sign messages; simply run the message to be signed through a decryption with your private key key. Anyone receiving the message can encrypt it with your publicly available key and read the message. Some algorithms do only one thing, others can both encrypt and authenticate.

The currently available public-key algorithms are listed in the following table:

Algorithm	Capabilities
RSA	Encryption, authentication/signatures
ElGamal	Encryption, authentication/signatures
DSA	Authentication/signatures

Many of these algorithms are patented. Before using any of them in a commercial product, consult a patent attorney; you may have to arrange a license with the patent holder.

An example of using the RSA module to sign a message:

```
>>> from Crypto.Hash import MD5
>>> from Crypto.PublicKey import RSA
>>> from Crypto import Random
>>> rng = Random.new().read
>>> RSAkey = RSA.generate(2048, rng)   # This will take a while...
>>> hash = MD5.new(plaintext).digest()
>>> signature = RSAkey.sign(hash, rng)
>>> signature   # Print what an RSA sig looks like--you don't really care.
('\021\317\313\336\264\315' ...,)
>>> RSAkey.verify(hash, signature)     # This sig will check out
1
>>> RSAkey.verify(hash[:-1], signature)# This sig will fail
0
```

Public-key modules make the following functions available:

construct(tuple): Constructs a key object from a tuple of data. This is algorithm-specific; look at the source code for the details. (To be documented later.)

generate(size, randfunc, progress_func=None, e=65537): Generate a fresh public/private key pair. `size` is a algorithm-dependent size parameter, usually measured in bits; the larger it is, the more difficult it will be to break the key. Safe key sizes vary from algorithm to algorithm; you'll have to research the question and decide on a suitable key size for your application. An N-bit keys can encrypt messages up to N-1 bits long.

`randfunc` is a random number generation function; it should accept a single integer `N` and return a string of random data `N` bytes long. You should always use a cryptographically secure random number generator, such as the one defined in the `Crypto.Random` module; **don't** just use the current time and the `random` module.

`progress_func` is an optional function that will be called with a short string containing the key parameter currently being generated; it's useful for interactive applications where a user is waiting for a key to be generated.

`e` is the public RSA exponent, and must be an odd positive integer. It is typically a small number with very few ones in its binary representation. The default value 65537 (=0b10000000000000001) is a safe choice: other common values are 5, 7, 17, and 257. Exponent 3 is also widely used, but it requires very special care when padding the message.

If you want to interface with some other program, you will have to know the details of the algorithm being used; this isn't a big loss. If you don't care about working with non-Python software, simply use the `pickle` module when you need to write a key or a signature to a file. It's portable across all the architectures that Python supports, and it's simple to use.

In case interoperability were important, RSA key objects can be exported and imported in two standard formats: the DER binary encoding specified in PKCS#1 (see RFC3447) or the ASCII textual encoding specified by the old Privacy Enhanced Mail services (PEM, see RFC1421).

The RSA module makes the following function available for importing keys:

importKey(externKey): Import an RSA key (pubic or private) encoded as a string `externKey`. The key can follow either the PKCS#1/DER format (binary) or the PEM format (7-bit ASCII).

For instance: >>> from Crypto.PublicKey import RSA >>> f = open("mykey.pem") >>> RSAkey = RSA.importKey(f.read()) >>> if RSAkey.has_private(): print "Private key"

Every RSA object supports the following method to export itself:

exportKey(format='PEM'): Return the key encoded as a string, according to the specified `format`: `'PEM'` (default) or `'DER'` (also known as PKCS#1).

For instance: >>> from Crypto.PublicKey import RSA >>> from Crypto import Random >>> rng = Random.new().read >>> RSAkey = RSA.generate(1024, rng) >>> f = open("keyPrivate.der","w+") >>> f.write(RSAkey.exportKey("DER")) >>> f.close() >>> f = open("keyPublic.pem","w+") >>> f.write(RSAkey.publickey().exportKey("PEM")) >>> f.close()

Public-key objects always support the following methods. Some of them may raise exceptions if their functionality is not supported by the algorithm.

can_blind(): Returns true if the algorithm is capable of blinding data; returns false otherwise.

can_encrypt(): Returns true if the algorithm is capable of encrypting and decrypting data; returns false otherwise. To test if a given key object can encrypt data, use `key.can_encrypt() and key.has_private()`.

can_sign(): Returns true if the algorithm is capable of signing data; returns false otherwise. To test if a given key object can sign data, use `key.can_sign() and key.has_private()`.

decrypt(tuple): Decrypts `tuple` with the private key, returning another string. This requires the private key to be present, and will raise an exception if it isn't present. It will also raise an exception if `string` is too long.

encrypt(string, K): Encrypts `string` with the private key, returning a tuple of strings; the length of the tuple varies from algorithm to algorithm. `K` should be a string of random data that is as long as possible. Encryption does not require the private key to be present inside the key object. It will raise an exception if `string` is too long. For ElGamal objects, the value of `K` expressed as a big-endian integer must be relatively prime to `self.p-1`; an exception is raised if it is not. Python 3.x: `‘string‘` must be an object interpretable as a buffer of bytes.

has_private(): Returns true if the key object contains the private key data, which will allow decrypting data and generating signatures. Otherwise this returns false.

publickey(): Returns a new public key object that doesn't contain the private key data.

sign(string, K): Sign `string`, returning a signature, which is just a tuple; in theory the signature may be made up of any Python objects at all; in practice they'll be either strings or numbers. `K` should be a string of random data that is as long as possible. Different algorithms will return tuples of different sizes. `sign()` raises an exception if `string` is too long. For ElGamal objects, the value

of `K` expressed as a big-endian integer must be relatively prime to `self.p-1`; an exception is raised if it is not. Python 3.x: `'string'` must be an object interpretable as a buffer of bytes.

size(): Returns the maximum size of a string that can be encrypted or signed, measured in bits. String data is treated in big-endian format; the most significant byte comes first. (This seems to be a **de facto** standard for cryptographical software.) If the size is not a multiple of 8, then some of the high order bits of the first byte must be zero. Usually it's simplest to just divide the size by 8 and round down.

verify(string, signature): Returns true if the signature is valid, and false otherwise. `string` is not processed in any way; `verify` does not run a hash function over the data, but you can easily do that yourself. Python 3.x: `'string'` must be an object interpretable as a buffer of bytes.

The ElGamal and DSA algorithms

For RSA, the `K` parameters are unused; if you like, you can just pass empty strings. The ElGamal and DSA algorithms require a real `K` value for technical reasons; see Schneier's book for a detailed explanation of the respective algorithms. This presents a possible hazard that can inadvertently reveal the private key. Without going into the mathematical details, the danger is as follows. `K` is never derived or needed by others; theoretically, it can be thrown away once the encryption or signing operation is performed. However, revealing `K` for a given message would enable others to derive the secret key data; worse, reusing the same value of `K` for two different messages would also enable someone to derive the secret key data. An adversary could intercept and store every message, and then try deriving the secret key from each pair of messages.

This places implementors on the horns of a dilemma. On the one hand, you want to store the `K` values to avoid reusing one; on the other hand, storing them means they could fall into the hands of an adversary. One can randomly generate `K` values of a suitable length such as 128 or 144 bits, and then trust that the random number generator probably won't produce a duplicate anytime soon. This is an implementation decision that depends on the desired level of security and the expected usage lifetime of a private key. I can't choose and enforce one policy for this, so I've added the `K` parameter to the `encrypt` and `sign` methods. You must choose `K` by generating a string of random data; for ElGamal, when interpreted as a big-endian number (with the most significant byte being the first byte of the string), `K` must be relatively prime to `self.p-1`; any size will do, but brute force searches would probably start with small primes, so it's probably good to choose fairly large numbers. It might be simplest to generate a prime number of a suitable length using the `Crypto.Util.number` module.

Security Notes for Public-key Algorithms

Any of these algorithms can be trivially broken; for example, RSA can be broken by factoring the modulus n into its two prime factors. This is easily done by the following code:

```
for i in range(2, n):
if (n%i)==0:
    print i, 'is a factor'
    break
```

However, `n` is usually a few hundred bits long, so this simple program wouldn't find a solution before the universe comes to an end. Smarter algorithms can factor numbers more quickly, but it's still possible to choose keys so large that they can't be broken in a reasonable amount of time. For ElGamal and DSA, discrete logarithms are used instead of factoring, but the principle is the same.

Safe key sizes depend on the current state of number theory and computer technology. At the moment, one can roughly define three levels of security: low-security commercial, high-security commercial, and military-grade. For RSA, these three levels correspond roughly to 768, 1024, and 2048-bit keys.

When exporting private keys you should always carefully ensure that the chosen storage location cannot be accessed by adversaries.

Crypto.Util: Odds and Ends

This chapter contains all the modules that don't fit into any of the other chapters.

Crypto.Util.number

This module contains various number-theoretic functions.

GCD(x,y): Return the greatest common divisor of `x` and `y`.

getPrime(N, randfunc): Return an `N`-bit random prime number, using random data obtained from the function `randfunc`. `randfunc` must take a single integer argument, and return a string of random data of the corresponding length; the `get_bytes()` method of a `RandomPool` object will serve the purpose nicely, as will the `read()` method of an opened file such as `/dev/random`.

getStrongPrime(N, e=0, false_positive_prob=1e-6, randfunc=None): Return a random strong `N`-bit prime number. In this context p is a strong prime if p-1 and p+1 have at least one large prime factor. `N` should be a multiple of 128 and > 512.

If `e` is provided the returned prime p-1 will be coprime to `e` and thus suitable for RSA where e is the public exponent.

The optional `false_positive_prob` is the statistical probability that true is returned even though it is not (pseudo-prime). It defaults to 1e-6 (less than 1:1000000). Note that the real probability of a false-positive is far less. This is just the mathematically provable limit.

`randfunc` should take a single int parameter and return that many random bytes as a string. If randfunc is omitted, then `Random.new().read` is used.

getRandomNBitInteger(N, randfunc): Return an `N`-bit random number, using random data obtained from the function `randfunc`. As usual, `randfunc` must take a single integer argument and return a string of random data of the corresponding length.

getRandomNBitInteger(N, randfunc): Return an `N`-bit random number, using random data obtained from the function `randfunc`. As usual, `randfunc` must take a single integer argument and return a string of random data of the corresponding length.

inverse(u, v): Return the inverse of `u` modulo `v`.

isPrime(N): Returns true if the number `N` is prime, as determined by a Rabin-Miller test.

Crypto.Random

For cryptographic purposes, ordinary random number generators are frequently insufficient, because if some of their output is known, it is frequently possible to derive the generator's future (or past) output. Given the generator's state at some point in time, someone could try to derive any keys generated using it. The solution is to use strong encryption or hashing algorithms to generate successive data; this makes breaking the generator as difficult as breaking the algorithms used.

Understanding the concept of **entropy** is important for using the random number generator properly. In the sense we'll be using it, entropy measures the amount of randomness; the usual unit is in bits. So, a single random bit has an entropy of 1 bit; a random byte has an entropy of 8 bits. Now consider a one-byte field in a database containing a person's sex, represented as a single character `'M'` or `'F'`. What's the entropy of this field? Since there are only two possible values, it's not 8 bits, but one; if you were trying to guess the value, you wouldn't have to bother trying `'Q'` or `'@'`.

Now imagine running that single byte field through a hash function that produces 128 bits of output. Is the entropy of the resulting hash value 128 bits? No, it's still just 1 bit. The entropy is a measure of how many possible states of the data exist. For English text, the entropy of a five-character string is not 40 bits;

it's somewhat less, because not all combinations would be seen. `'Guido'` is a possible string, as is `'In th'`; `'zJwvb'` is not.

The relevance to random number generation? We want enough bits of entropy to avoid making an attack on our generator possible. An example: One computer system had a mechanism which generated nonsense passwords for its users. This is a good idea, since it would prevent people from choosing their own name or some other easily guessed string. Unfortunately, the random number generator used only had 65536 states, which meant only 65536 different passwords would ever be generated, and it was easy to compute all the possible passwords and try them. The entropy of the random passwords was far too low. By the same token, if you generate an RSA key with only 32 bits of entropy available, there are only about 4.2 billion keys you could have generated, and an adversary could compute them all to find your private key. See RFC 1750, "Randomness Recommendations for Security", for an interesting discussion of the issues related to random number generation.

The `Random` module builds strong random number generators that look like generic files a user can read data from. The internal state consists of entropy accumulators based on the best randomness sources the underlying operating is capable to provide.

The `Random` module defines the following methods:

new(): Builds a file-like object that outputs cryptographically random bytes.

atfork(): This methods has to be called whenever os.fork() is invoked. Forking undermines the security of any random generator based on the operating system, as it duplicates all structures a program has. In order to thwart possible attacks, this method shoud be called soon after forking, and before any cryptographic operation.

get_random_bytes(num): Returns a string containing `num` bytes of random data.

Objects created by the `Random` module define the following variables and methods:

read(num): Returns a string containing `num` bytes of random data.

close(): **flush()**: Do nothing. Provided for consistency.

Crypto.Util.RFC1751

The keys for private-key algorithms should be arbitrary binary data. Many systems err by asking the user to enter a password, and then using the password as the key. This limits the space of possible keys, as each key byte is constrained within the range of possible ASCII characters, 32-127, instead of the whole 0-255 range possible with ASCII. Unfortunately, it's difficult for humans to remember 16 or 32 hex digits.

One solution is to request a lengthy passphrase from the user, and then run it through a hash function such as SHA1 or MD5. Another solution is discussed in RFC 1751, "A Convention for Human-Readable 128-bit Keys", by Daniel L. McDonald. Binary keys are transformed into a list of short English words that should be easier to remember. For example, the hex key EB33F77EE73D4053 is transformed to "TIDE ITCH SLOW REIN RULE MOT".

key_to_english(key): Accepts a string of arbitrary data `key`, and returns a string containing uppercase English words separated by spaces. `key`'s length must be a multiple of 8.

english_to_key(string): Accepts `string` containing English words, and returns a string of binary data representing the key. Words must be separated by whitespace, and can be any mixture of uppercase and lowercase characters. 6 words are required for 8 bytes of key data, so the number of words in `string` must be a multiple of 6.

Extending the Toolkit

Preserving a common interface for cryptographic routines is a good idea. This chapter explains how to write new modules for the Toolkit.

The basic process is as follows:

1. Add a new `.c` file containing an implementation of the new algorithm. This file must define 3 or 4 standard functions, a few constants, and a C `struct` encapsulating the state variables required by the algorithm.

2. Add the new algorithm to `setup.py`.

3. Send a copy of the code to me, if you like; code for new algorithms will be gratefully accepted.

Adding Hash Algorithms

The required constant definitions are as follows:

```
#define MODULE_NAME MD2     /* Name of algorithm */
#define DIGEST_SIZE 16          /* Size of resulting digest in bytes */
```

The C structure must be named `hash_state`:

```
typedef struct {
 ... whatever state variables you need ...
} hash_state;
```

There are four functions that need to be written: to initialize the algorithm's state, to hash a string into the algorithm's state, to get a digest from the current state, and to copy a state.

- `void hash_init(hash_state *self);`
- `void hash_update(hash_state *self, unsigned char *buffer, int length);`
- `PyObject *hash_digest(hash_state *self);`
- `void hash_copy(hash_state *source, hash_state *dest);`

Put `#include "hash_template.c"` at the end of the file to include the actual implementation of the module.

Adding Block Encryption Algorithms

The required constant definitions are as follows:

#define MODULE_NAME AES /* Name of algorithm */ #define BLOCK_SIZE 16 / Size of encryption block / #define KEY_SIZE 0 / Size of key in bytes (0 if not fixed size) */

The C structure must be named `block_state`:

```
typedef struct {
 ... whatever state variables you need ...
} block_state;
```

There are three functions that need to be written: to initialize the algorithm's state, and to encrypt and decrypt a single block.

- `void block_init(block_state *self, unsigned char *key, int keylen);`
- `void block_encrypt(block_state *self, unsigned char *in, unsigned char *out);`
- `void block_decrypt(block_state *self, unsigned char *in, unsigned char *out);`

Put `#include "block_template.c"` at the end of the file to include the actual implementation of the module.

Adding Stream Encryption Algorithms

The required constant definitions are as follows:

```
#define MODULE_NAME ARC4       /* Name of algorithm */
#define BLOCK_SIZE 1           /* Will always be 1 for a stream cipher */
#define KEY_SIZE 0             /* Size of key in bytes (0 if not fixed size) */
```

The C structure must be named `stream_state`:

```
typedef struct {
 ... whatever state variables you need ...
} stream_state;
```

There are three functions that need to be written: to initialize the algorithm's state, and to encrypt and decrypt a single block.

- `void stream_init(stream_state *self, unsigned char *key, int keylen);`
- `void stream_encrypt(stream_state *self, unsigned char *block, int length);`
- `void stream_decrypt(stream_state *self, unsigned char *block, int length);`

Put `#include "stream_template.c"` at the end of the file to include the actual implementation of the module.