Information Security
P1 Crypto-Misuse Challenge v1.1

Winter term 2019/2020

Introduction

There are many pitfalls when using cryptographic algorithms. Most of these pitfalls do not come from the basic building blocks, such as the Advanced Encryption Standard (AES); they are secure when implemented correctly. Instead, most of the pitfalls come from the incorrect usage of these blocks, often by inexperienced engineers.

In this challenge, you have to exploit some of the most common mistakes. And bear in mind: **All these things happened!** The concrete scenarios are constructed, but the underlying errors appeared countless times, not only in some hobbyist project but often in high-profile applications.

Framework

There are a total of 7 challenges. The number of points awarded for each challenge is stated in this document. If you have any questions, please contact the responsible teaching assistants: `lena.heimberger@student.tugraz.at`, `haubenwallner@student.tugraz.at`, or ask a question in the newsgroup: `tu-graz.lv.infosec`.

The challenges are written and have to be solved in Python (Python3). The following packages must be installed, use, e.g., `pip install <package_name> --user`

- pycryptodomex
- ecdsa

You get two archives, one contains code, the other challenge files. They have the same file structure, unpack them in the same location (i.e., unpack the challenge files in your code directory). Each challenge is contained in a subdirectory, the concrete folder is stated for each challenge in this document. After unpacking both archives, each challenge folder contains the following:

- Python source file (`<folder_name>.py`), sometimes with additional source files
- `challenge*` file(s), such as ciphertexts that you need to decrypt. The names of these files can vary.
- `*_solution` file, which allows you to test your implementation

Each challenge script offers a small command-line interface, type "`python3 <challenge_name>.py --help`" to get an overview of the implemented sub-commands. Many scripts offer an encryption/decryption functionality, which can be invoked via "`python3 <challenge_name>.py e file_list`" and "`python3 <challenge_name>.py d file_list`", respectively. All challenge files were generated using the provided scripts. Thus, you should have a look at how things are implemented there and can also generate new challenges yourself.

You can run the challenges by calling "`python3 <challenge_name>.py c`". By default, this tries to solve the provided challenges, but you can specify your own ones by simply giving a file list. This calls the function `solve_challenge`, which you have to extend such that it solves the challenge. **Put all your code in the specified block** (you can also add new sub-routines and include standard Python packages). Immediately before the specified block, a secret variable is initialized to a dummy value. **Your code has to recover the value of this secret variable**.

Finally, the outcome is compared to the provided solution and the script will tell you if you solved the challenge correctly. For testing, we will run your program with fresh challenges. On our test system, the maximum execution time is 10 seconds per challenge, for challenge `textbook_rsa` you have 2 minutes.

1 The ECB Mode of Operation (3P)

Folder: `ecb`

Block ciphers operate on data blocks of fixed size, typically 128 bits. You usually want to encrypt data of arbitrary size; the easiest way to achieve this is to simply cut the data into blocks and encrypt them individually (Figure 1). This is called the Electronic Code Book (ECB) mode.

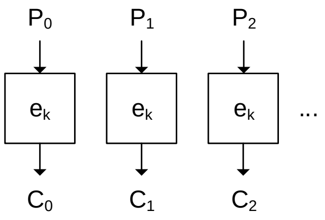


Figure 1: The ECB mode of operation

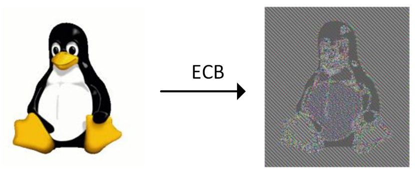


Figure 2: The problem with ECB

However, there is a major problem with ECB: same input blocks always lead to the same output blocks. The downsides of this can be seen in Figure 2. White patches always encrypt to some ciphertext c_{white}, black patches to a different ciphertext c_{black}. This makes it very easy to spot patterns and in this case still recognize the image.

Scenario. A hardware vendor not well versed in security produces a wireless keyboard. As he wants to prevent sniffing of the input, he encrypts the communication but does a bad job at it. Since there cannot be any delays when sending a keystroke to the PC, he takes the code for the keystroke, pads it to 128 bits, then encrypts it (in ECB mode), and finally sends it the ciphertext to the PC. For this challenge, we make a simplification and do not look at individual keystrokes, but have an ASCII text that is encrypted character-wise.

Challenge. Bruce[1] is an avid blogger and uses the insecure keyboard described above. You are able to get in his vicinity and sniff the encrypted communication between keyboard and PC. Around that time, you note that one of his blog posts goes online, so you can assume that some of the sniffed ciphertext corresponds to the text of the blog post. You can also assume that he has to login to his site before he can submit a post. Retrieve his password!

You can use the following assumptions:

- His username is `bruce`. His password can be found between his username and the first letter of the blog post.
- All characters found in his password also appear in the blog post.
- He typed in the blog post in one piece, without any corrections and other inputs happening in this time. In other words: some part of the ciphertext directly corresponds to the text.
- You do not know when exactly the text was typed in and have to assume that before and afterward other things, not related to the blog, were typed in. That is, you have to determine which part of the ciphertext corresponds to the blog post.

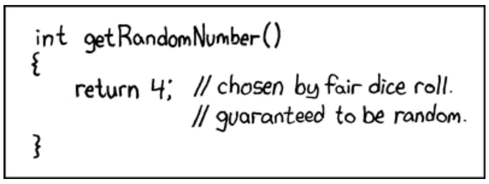
Hint: for finding the correct offset, focus on one specific character (e.g., all the spaces in the text)

[1]`https://www.schneier.com/`

2 Nonce Reuse (2P + 2P)

There do exist secure alternatives to ECB, in which encrypting the same plaintext block does not result in the same ciphertext block. Examples are the Counter Mode (CTR) and Cipher Block Chaining (CBC), but you can also use stream ciphers. These alternatives, however, require initialization with a random number, the so-called Initialization Vector (IV), often also called the "nonce" (number used once). The second name already implies that you have to use a fresh nonce each time you encrypt something, reusing the nonce (or, at least in some settings, using a related nonce) can have drastic consequences.

Still, there have been many instances where the nonce was reused, e.g., by using a random number generator similar to the one shown in Figure 3.



```
int getRandomNumber()
{
    return 4;  // chosen by fair dice roll.
               // guaranteed to be random.
}
```

Figure 3: The XKCD random number generator (`https://xkcd.com/221/`)

2.1 Symmetric Cryptography (2P)

Folder: `nonce_reuse_sym`

For nonce reuse in the symmetric-key setting, we focus on a stream cipher. A stream cipher takes as input a secret key and a random initialization value (IV), and then generates an arbitrarily long *keystream*, which is XORed ($\oplus$) to the plaintext to receive the ciphertext. When **using the same IV twice** (for the same key), then the **same keystream** is generated. In other words, the stream cipher then behaves like a **reused One-Time Pad**.

Scenario A car manufacturer uses a keyless entry system based on **rolling codes**. In a rolling code[2], both the key and the car share a common secret state. Upon pressing a button on the key fob, the key updates the state in some manner and then securely sends this updated state. Upon receiving the command, the car updates its own local state in the same way and then checks if the received state matches the local result. As keys can be pressed accidentally (without the car receiving the message), the car actually computes multiple state updates, checks if the received state matches any of them, and then advances the state accordingly. Commonly, the shared secret state is simply a counter, which is sent encrypted using a shared secret key.[3]

The system you are supposed to attack uses an $n = 64$-bit **linear feedback shift register (LFSR)** instead of a simple counter. Figure 4 shows the structure of such an LFSR. The positions of the XORs are known (for instance, through reverse engineering) and are included in the script. The state, however, is secret. For each button press, the current state of the LFSR is encrypted using a stream cipher and then sent, and finally the LFSR is clocked n times to update the state.

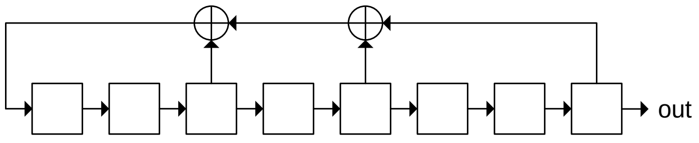


Figure 4: An $n = 8$-bit linear feedback shift register (LFSR). Bits are represented as squares, the rightmost bit is used as output.

[2]`https://en.wikipedia.org/wiki/Rolling_code`

[3]`https://www.usenix.org/system/files/conference/usenixsecurity16/sec16_paper_garcia.pdf`

Challenge. You find out, that the manufacturer did not include a random-number generator on the key and thus **always uses the same IV** for the stream cipher. Then, by placing a receiving device near a car you want to steal, you can sniff many messages sent from a real key. Your task is to **predict the next message** sent by the key, and thus be able to unlock the car while the owner is not around.

Hints. LFSRs consist solely of XORs. Thus, they are **linear** in this regard. This means that when having two LFSR states a, b, then $\mathrm{LFSR}(a \oplus b) = \mathrm{LFSR}(a) \oplus \mathrm{LFSR}(b)$.

This happened. The KRACK attack[4] was able to break the security of WPA2, which is used to secure WIFI (WLAN). By carefully manipulating and replaying messages, an attacker can achieve that the key is "reinstalled" (KRACK: Key Reinstallation Attack). Reinstalling the key also means that the counter is reset to its initial value. Thus the nonce is reused. WPA2 uses the counter mode for encryption. Hence, the exploitation is similar to the one in the challenge.

2.2 Asymmetric Cryptography (2P)

Folder: `nonce_reuse_asym`

It's not only symmetric-key cryptography which requires randomness, but most secure asymmetric (public-key) schemes also require a nonce. One example is the Elliptic Curve Digital Signature Standard (ECDSA). The algorithm is now briefly described. For solving the challenge, no knowledge about elliptic curve-arithmetic is required. You do however need some discrete maths.

ECDSA is now briefly explained. For the challenge, you can ignore all parts apart from line 6, which contains a simple modular equation.

In ECDSA, one has public parameters G (a point on the elliptic curve) and n (the group order). The private key d_A is a random number (integer) in the range $[1, n-1]$. The public key $Q_A = d_A \times G$. Signing works as follows:

Algorithm 1 ECDSA Signing Algorithm

```
Input: Message $m$, private key $d_A$, public parameters $(n, G)$
Output: Signature $(r, s)$
1: $e = \mathsf{HASH}(m)$
2: $z =$ leftmost $n'$ bits of $e$, with $n'$ the bit length of $n$
3: Select a random integer $k$ in the range $[1, n-1]$.
4: Compute $(x_1, y_1) = k \times G$                    ▷ Point-Scalar Multiplication
5: $r = x_1 \bmod n$
6: $s = k^{-1}(z + r \cdot d_A) \bmod n$
7: return $(r, s)$
```

As a hint, have a look at the description of ECDSA in Wikipedia, which also discusses what can happen when you reuse the nonce k.[5]

Challenge. You are given two messages and their according signatures. To save on randomness, only the first nonce is generated randomly. The nonce for the second signature is derived from the previous nonce as $k = a \cdot k + b \bmod n$, with $(a = 2, b = 1)$. Recover the private key d_A.

This happened. Sony used a constant nonce for signing firmware packages of their PlayStation3 console. This allowed simple recovery of the signing key.[6]

[4] `https://www.krackattacks.com/`
[5] `https://en.wikipedia.org/wiki/Elliptic_Curve_Digital_Signature_Algorithm`
[6] `https://www.bbc.co.uk/news/technology-12116051`

3 Encryption without Authentication (3P)

Folder: `enc_without_auth`

Encryption provides confidentiality, which means that nobody can recover the message without having the key. It, however, does not provide integrity/authenticity. In other words, you do not know if the ciphertext was truly generated by someone having the key, or if it is just random or specially crafted data. This is easy to see for stream ciphers, where the ciphertext c is the XOR of a keystream k and the plaintext m: $c = m \oplus k$. When the attacker intercepts the ciphertext, flips one bit in it, and then forwards it, the receiver will after decryption have the same bit-flip in the plaintext, and have no ability to detect this flip. Other encryption modes, such as CBC, suffer from the same fate, although exploitation might not be as straightforward.

Challenge. A company uses contactless smart-cards for access control (opening doors). The company has **two security domains**: standard cards can only open the front door, whereas high-security cards can only unlock the highly confidential lab. You are able to install a sniffing device on the front door, and can thus intercept as many protocol executions as you want. You are also in possession of a portable device allowing the injection of packets containing data of your choosing. Your goal is to **open the lab door**.

The system uses the *challenge-response* protocol shown in Figure 5. For encryption, all parties (cards and scanners for both security domains) share the **same key secret key** k (which you do not know). In this protocol, the two parties prove knowledge of the shared secret key to each other using a challenge-response approach. The card chooses a random **challenge** n_C, sends n_C in plain (unencrypted) to the scanner, who then returns the encryption of the challenge, a.k.a., the **response** $E_k(n_C)$. The card then tests if the decryption of the response matches the sent challenge. The same is repeated in the other direction.

On top of that, all cards and scanners keep lists of authorized devices. That is, each scanner keeps a list of authorized cards (their IDs), and refuses to talk to unknown IDs; the same applies for cards. Thus, a low-security card will not receive an answer from the high-security scanner, even though they share the same key. However, you know all IDs through a database leak.

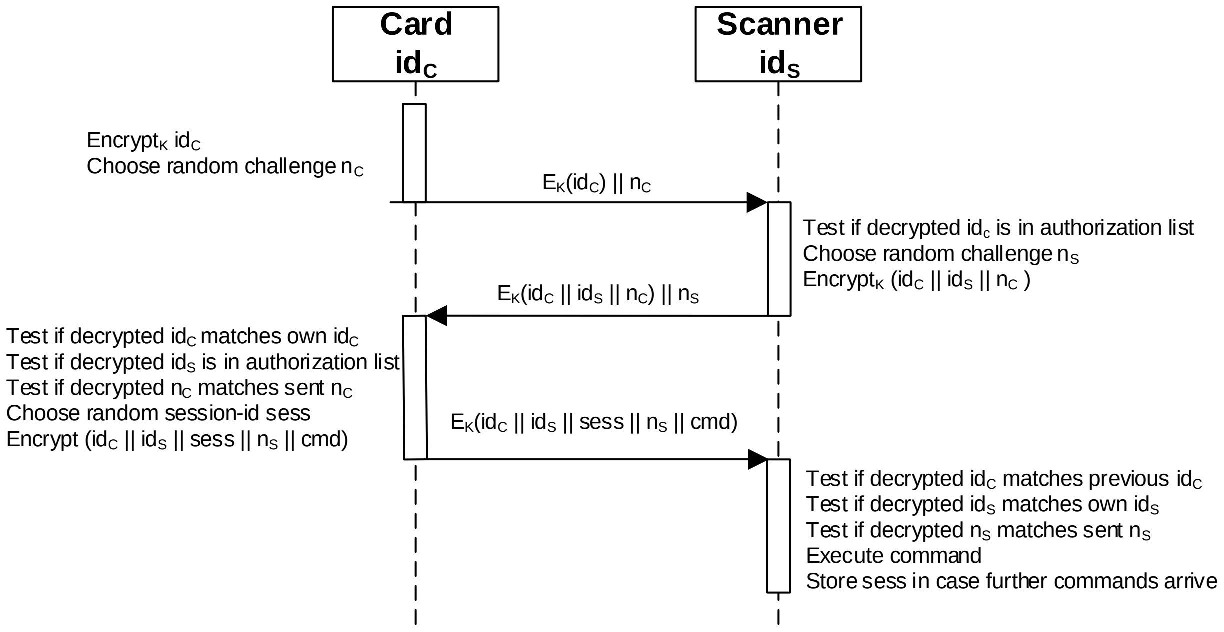

Card
id_C
Scanner
id_S
$Encrypt_K$ id_C
Choose random challenge n_C
$E_K(id_C) \| n_C$
Test if decrypted id_C is in authorization list
Choose random challenge n_S
$Encrypt_K$ $(id_C \| id_S \| n_C)$
$E_K(id_C \| id_S \| n_C) \| n_S$
Test if decrypted id_C matches own id_C
Test if decrypted id_S is in authorization list
Test if decrypted n_C matches sent n_C
Choose random session-id sess
Encrypt $(id_C \| id_S \| sess \| n_S \| cmd)$
$E_K(id_C \| id_S \| sess \| n_S \| cmd)$
Test if decrypted id_C matches previous id_C
Test if decrypted id_S matches own id_S
Test if decrypted n_S matches sent n_S
Execute command
Store sess in case further commands arrive

Figure 5: The used challenge-response protocol

A random session-id `sess` is also generated by the card, it allows sending subsequent commands that should be linked to the first, if needed. The command `cmd` to open any door is simply the ASCII-encoding of "opendoor".

All encryptions are performed using AES-128 in CBC mode. The plaintext is first padded using PKCS7 padding, the IV used for CBC is prepended to the ciphertext. That is, encrypted packets have the form: `[IV || CBC-Enc(Pad(Data))]`. No data authentication is used. All IDs are 64 bits long, challenges and the session-id are 128 bits long.

Hints. You can break the system by exploiting that data is encrypted but not authenticated. That is, you can manipulate the ciphertexts without being detected (if you do it properly). You can, for example, rearrange message blocks, replace IVs, and flip bits in blocks. Your goal is to craft valid packets using only attainable information (sniffed packets, responses to proper packets, responses to injected packets, etc.).

Try to draw the CBC decryption process of the targeted packet and think of ways how to acquire all the needed information.

4 Bad Randomness (2P + 2P)

Most cryptographic protocols (like the ones above) require some sort of randomness coming from a random number generator (RNG). However, there are some important requirements that the RNG must fulfill in order to allow security. For instance, when you get any number of bits produced by the RNG, e.g., as part of the public nonce, then you should still not be able to predict what the RNG will output next, or what it has output before. RNGs having this property are called Cryptographically Secure Pseudo-Random Number Generator (CSPRNGs) (“pseudo” because the process is deterministic and depends on a seed value). Using non-cryptographic RNGs or improper initialization of CSPRNGs in a cryptographic context can have a catastrophic impact.

Scenario. You are given a file-encryption tool. This tool takes an input file, generates a random IV and key, encrypts the file content, and finally stores the ciphertext and the encryption key in separate files. This way, you can upload your files to some untrusted storage provider and keep the keys on your local computer (Note: there are much better solutions for this problem).

A new random key is generated for each file, but the used RNG is flawed. Recover the key used to encrypt the challenges, and then decrypt the challenges!

There are two implementations having different flaws. Both generate the IV before generating the key. The IV is included as the first 16 bytes of the encrypted file, the remainder of the file contains the actual encrypted content.

4.1 Insecure RNGs (2P)

Folder: `bad_rand_rng`

The first implementation uses a re-implementation of C’s `rand()` function. The concrete implementation of this function is up to the developers of the used C standard library. For this challenge, we use the linear-congruential generator included in glibc. It has an internal 32-bit state `next`. When calling the RNG, the state is updated using:

```
next = ((next * 1103515245) + 12345) & 0x7fffffff
```

Then, the updated `next` is returned as the random number.

Challenge. Recover the plaintext of the challenge file! Hint: observe that the IV is public and generated before the key.

4.2 Insecure Initialization (2P)

Folder: `bad_rand_seed`

CSPRNGs are not random algorithms, but entirely deterministic. Thus, they need to be initialized with a truly random value, the so-called seed. Using the same seed twice will result in the exact same output. Generating such a truly random seed is not a trivial task, especially on smaller devices such as microcontrollers. It is also easy to make mistakes in this generation.

This second implementation of the file encryption tool uses a proper CSPRNG but makes an error in the generation of the seed.

Challenge. Find the error in the seed generation and recover the plaintext of the challenge file! Hint: use side-channel information. The OS and filesystem stores more about a file than just its contents.

5 Textbook RSA (2P)

Folder: `textbook_rsa`

The most simple variant of RSA is called "textbook RSA", because this version is often presented in textbooks (german: Lehrbücher). When you want to encrypt a message m with a public key (e, n), you compute $c = m^e \bmod n$, for decryption with the private key d you compute $m = c^d \bmod n$. This straight-forward approach has two undesirable properties:

1: Malleability. The ciphertext is malleable (german: "formbar"), which means you can manipulate it to something related without knowing the plaintext. Example: you intercept a ciphertext c, compute $c \cdot 2^e \bmod n$, and forward this. The receiver will decrypt this to $2m$, which means you altered the plaintext in some known way.

2: Determinism. Encrypting the same m twice will lead to the same c.

Challenge. You are given an RSA public key and a ciphertext. You happen to know that the corresponding message is a 3-letter English word ([a-z], all lowercase). Recover the plaintext!

Hint: use one of the undesirable properties. Do not try to recover the key.

This happened. In the Chinese QQ browser (see Section 7.2).

6 Version History

6.1 v1.1

Fixed wrong values for a and b in the `nonce_reuse_asym` assignment. The new values of $a = 2, b = 1$ now reflect the numbers used in the code.

7 Further Reading

The above mistakes are some of the most common ones, but by far not the only ones you can make. Some more errors are now given, as well as methods to prevent them without requiring in-depth knowledge of cryptography. This section is not required for solving the challenges, but reading still recommended.

7.1 Other Mistakes

The article “Top 10 Developer Crypto Mistakes”[7] gives a nice overview of many errors. Some of the ones not covered in the challenges are:

Hard-coded keys. When secret keys are included in a binary that is then distributed (via download, in a firmware that can be read out, etc.), it is easy to recover the key from this binary.

No key diversity. If the same symmetric key is used by a large number of similar devices, then recovering the key just once allows attacks on all devices. This makes, e.g., invasive attacks much more lucrative. As an example, the remote keyless entry system of VW used the same symmetric key for all shipped cars and their keys for many years. After extracting this key once, researchers were able to unlock a vast number of cars.[8]

Outdated cryptography. Many applications still use outdated and insecure cryptographic algorithms. For instance, RSA keys should nowadays be at least 2048 bits long, but 1024-bit and even 512-bit keys can still be found. An even more extreme case was given in Section 5. For hash functions, MD5 and SHA-1 should not be used anymore. The use of MD5 allowed a group to create forged digital certificates.[9]

Passwords, passwords, passwords. Passwords are a very sensitive topic where many things can go wrong. On servers, passwords should never be stored in plain text. Otherwise, an attacker having access to the server can simply read out all passwords. Storing a hash of a password is more secure, but can be defeated with so-called Rainbow Tables, which store the hashes of many “popular” passwords. The most secure variant is to use dedicated hash functions which also take as input a so-called salt. Similar things are also true for password-based key derivation.

Invent your own crypto / security by obscurity. Never roll/invent your own crypto! Toying around is, of course, fine, but never deploy it. This is also nicely captured by “Schneier’s Law”: *Anyone, from the most clueless amateur to the best cryptographer, can create an algorithm that he himself can’t break. It’s not even hard. What is hard is creating an algorithm that no one else can break, even after years of analysis. And the only way to prove that is to subject the algorithm to years of analysis by the best cryptographers around.*

The above already implies that keeping your algorithm secret (security by obscurity) is not a remedy. As soon as the algorithm is reverse-engineered or leaked, it will fall apart. A very prominent example of this is the CRYPTO1 algorithm used in Mifare Chipcards. As soon as the CRYPTO1 algorithm became public, it was broken. You could use that, e.g., to have free rides on the London Tube.[10]. Another very recent example is the proprietary crypto used by Tesla car keys, which can be broken in just 2 seconds.[11]

[7] `https://littlemaninmyhead.wordpress.com/2017/04/22/top-10-developer-crypto-mistakes/`
[8] `https://www.usenix.org/system/files/conference/usenixsecurity16/sec16_paper_garcia.pdf`
[9] Researchers Use PlayStation Cluster to Forge a Web Skeleton Key, `https://www.wired.com/2008/12/berlin/`
[10] `https://www.wired.com/2008/06/hackers-crack-l/`
[11] `https://www.engadget.com/2018/09/10/tesla-model-s-key-fob-cloning-vulnerability/`

7.2 The Ultimate Example

The Chinese Mobile Browser QQ is the ultimate amalgamation of the discussed flaws. It features[12]

- Hard-coded keys (Section 7.1)
- Textbook RSA (Section 5)
- An insecure RNG (Section 4.1) with an easy-to-guess initialization (Section 4.2)
- Outdated Cryptography and insufficient key lengths (Section 7.1). An earlier version used 128-bit RSA keys, which are trivial to factor. A newer version upgraded to 1024-bit keys, which also should not be used anymore.
- ECB mode (Section 1)

7.3 How To Avoid Mistakes

There are some simple rules to stay clear of the most basic mistakes.

- Never implement cryptographic algorithms on your own! Always use some tried and tested libraries. The only exceptions are educational purposes (but then never use it in a productive environment) or if you absolutely have to and know exactly what you are doing.
- Use misuse-resistant libraries. Good cryptographic libraries do not give the user access to low-level algorithms and thus simply do not allow the user to make the mistakes. This means, for instance, that textbook RSA is disabled, that it is not possible to have a user-defined nonce (the library chooses a nonce for you), that secure random-number generation is already in-built, or that only one (or a selected few) secure authenticated modes of operation are supported. Some examples of such libraries are NaCl (pronounced "salt")[13] and Google's Tink[14].

[12] When Textbook RSA is Used to Protect the Privacy of Hundreds of Millions of Users: `https://arxiv.org/pdf/1802.03367.pdf`

[13] `https://nacl.cr.yp.to/`

[14] `https://github.com/google/tink`