PassLok technical document

(updated as of version 2.4.2, 9/6/17)

By F. Ruiz and team.

In this document, we explain what drives the design of the different components and functions of PassLok, and the specific decisions made at every point. It is meant as a companion to the actual source code, in order to help those trying to audit it. Unless explicitly noted, everything here pertains to both the standalone app, PassLok Privacy, and its derivatives and to the Chrome and Firefox extension PassLok for Email. This is a document in progress.

General design criteria

PassLok is conceived as a lightweight, portable, comprehensive cryptographic and steganographic toolkit that can be used to supplement other programs. For instance, a user would have PassLok open alongside a conventional webmail client, and cut and paste between the two in order to provide end-to-end encryption for mail messages. Another user goes a step forward, and attaches image files that contain the encrypted messages in a way undetectable to others, which might be necessary in places where encryption use is criminalized.

Here is a list of design goals:

- High cryptographic strength.
- High portability. This means no installation, if possible.
- High cross-platform compatibility.
- Reasonable performance even on low-end devices such as smartphones.
- Inherently trustworthy. If the complete code is human-readable, so much better.
- Minimum reliance on servers. The idea is not to be forced to trust anyone.
- Very easy to use even for cryptography laypersons.
- Aimed at asynchronous communications, but can do synchronous as well if it is not too hard.

To meet these goals, PassLok consists essentially of HTML and JavaScript code, which has the following desirable traits:

- Great cross-platform compatibility, since HTML pages run on any device and any operating system with a decently modern browser available. Users also tend to be proficient with browsers, which enhances ease of use.
- Auditability: the entire code is human-readable, so expert users are able to scan it for possible vulnerabilities without resorting to specialized software.
- Authenticity. Once the code is trusted, it is fairly easy to make sure that no changes have been introduced, simply by comparing a one-way hash of it with a trusted value.

- Computations coded in JavaScript are performed locally, without involving a server that might be compromised. The code can also be executed without a network connection.

There are also a few difficulties introduced with this approach:

- HTML pages are usually not given full access to the computer hardware or the rest of the operating system. This means that interaction with other apps will require the manual use of the clipboard. File I/O will also be subjected to important restrictions.
- JavaScript is considered unsafe for cryptography, according to some experts. Their criticism is usually aimed at network interactions, however, which do not exist in PassLok.
- The code will run more slowly than if it had been converted to a lower-level language. Modern browsers, on the other hand, include highly optimized JavaScript engines so that in practice this is not a serious problem.

We now pass to discuss the different design aspects, beginning with those under the hood. But before we start, a warning for experts reading this document: in order to make it as accessible as possible to common users, PassLok uses certain words with a meaning that may be different from the commonly accepted meaning among experts, but which is believed to be closer to the meaning understood by common users. Whenever this situation is encountered, the word will be capitalized. Thus, an "Anonymous" message is not one that cannot be traced over a network, but rather one whose authorship cannot be decided from the massage itself. Similarly, a "Signed" message does not involve a digital signature, but rather it is devised so that it will not be decrypted unless the sender is correctly identified by means of his/her public key. A "Key" is a private key, and its matching "Lock" is its matching public key. Additional terms will be defined as they are encountered.

Cryptographic methods

PassLok implements symmetric encryption as well as three kinds of asymmetric encryption. The underlying crypto engine prior to version 2.2 was SJCL, an open-source JavaScript library originally developed by a team of researchers at Stanford University. With version 2.2.0, PassLok switched to TweetNaCl, a JavaScript implementation of the open-source NaCl suite, by Dan Bernstein et al. SJCL is a few years old and has reached a stable condition but it is still actively maintained. Tweet NaCl is newer but already tested against the original C-source NaCl. Both have been checked by experts and flaws are timely patched. SJCL is also quite speedy compared to similar codes, but NaCl is significantly faster in all its operations.

These are the features of SJCL that were used in PassLok up to version 2.1:

- Symmetric encryption/decryption using the Advanced Encryption Standard (AES, also known as Rijndael). SJCL allows 128-, 192-, and 256-bit keys; PassLok uses 256-bit keys exclusively. SJCL outputs in either CCM or OCB2 mode, both of which are authenticated modes. PassLok uses the default CCM mode in every instance. The SJCL engine is also capable of using as plaintext the full range of UTF16 characters, which is useful for non-western languages.
- Cryptographically strong pseudo-random number generation. SJCL has a timer-based entropy collection function, which is initialized as soon as PassLok loads. Additional entropy is collected every time the user interacts with an interface element, which happens at non-predictable times (especially the fractions of a second, which is what is used to collect entropy).
- Elliptic curve math. SJCL supports Weierstrass elliptic curves, of the form $y^2 = x^3 + a*x + b$, and includes the specific parameters for a few standard curves. PassLok uses the p521 curve over a prime field standardized by NIST. The p521 parameters were not originally included in the official SJCL distribution but were added by our team. These parameters were subsequently discussed by others on the SJCL forum, and there was a perfect match with our parameters. The p521 curve is based on the $2^{521} – 1$ Mersenne prime number, which has interesting properties that provide speed and space savings. After a pull request was submitted and accepted, the SJCL master now contains the parameters for this curve. We discuss later some more on this choice.
- SHA512 one-way hash, which is used in the signature function of PassLok.
- The SCRYPT key-generation algorithm, although not an official component of SJCL, has been developed by an open-source third party to blend seamlessly with the SJCL library. It is used in PassLok to convert user-provided keys (which tend to be short) into the random-looking 256-bit keys that are actually used to encrypt and decrypt in AES, and 521-bit private keys used in symmetric encryption. SCRYPT has been designed specifically to defeat attempts at building massively parallel key-guessing hardware.

With version 2.2, PassLok made the switch to NaCl, which is also open-source and has significant differences with respect to SJCL. The list above is repeated here for NaCl:

- Instead of AES, NaCl uses XSalsa20, a stream cipher that, although not standardized, has been proposed as a standard for the eSTREAM competition, which bears some similarities to the older AES competition but focuses on faster stream ciphers. In the roughly 10 years since XSalsa20 was developed, no significant attack has yet been found against it. The reason why XSalsa20 is selected is because of the convenience of using it together with the Curve25519 elliptic curve, by the same authors. XSalsa20 is significantly faster than AES (up to three times), but this is not a crucial consideration for PassLok.

- NaCl uses the PRNG built into most modern web browsers rather than having its own. This has the advantage that entropy collection is much more effective. The library gives an error if no secure PRNG is available.

- Two elliptic curves are built into NaCl: Curve25519, a Montgomery curve of the form $y^2 = x^3 + a*x^2 + x$, and Ed25519, a Twisted Edwards curve of the form $y^2 = x^2 + a*x^2*y^2 + 1$. The two curves are used for different purposes. Curve25519 is used for establishing Diffie-Hellman combinations of public and private key; Ed25519 is used for a digital signature scheme using the Schnorr algorithm. The two curves are related by a change of variable, so that a point in one corresponds to a point in the other. Using them is slightly more complex than with the curves in SJCL, but still manageable.

- NaCl uses SHA-512 only as a hash function.

- There is a JavaScript implementation of SCRYPT that matches the NaCl library quite well, although it does not depend on it. It is roughly twice as fast as the one used with SJCL, which permits a double number of iterations. It is by the same authors as the JavaScript TweetNaCl, and originally designed to run asynchronously, but the current version also supports synchronous operation, which is the way it is used in PassLok.

NaCl runs faster than SJCL, but this is not the reason why it was chosen for version 2.2 of PassLok. The real reason is that the additional strength of the p521 curve, which is currently supported only by SJCL, was seen as unnecessary in view of other components of the code that are not nearly as strong. If Curve25519 is adequate for the public key functions of PassLok, then it is preferred because of its lack of ties with the NSA, which is suspected by some to have fostered flawed standards in order to exploit them later on. Thus, there is a cloud of suspicion over the NIST-standardized curves, which were actually developed by the NSA, while there is none for Curve25519 and its derivatives. Below are the key arguments that led to the decision to switch from SJCL to NaCl:

- Both AES-256 and XSalsa20 have a computational complexity (number of calculations needed to reverse them) of around 250 bits, since some algorithms have been discovered that beat brute force search by a few bits, but no more. 250 bits is still considered much stronger than necessary given current computational power. So in this regard XSalsa20 has the edge because of its superior speed, though it is not as well-vetted as AES.

- Both Curve25519 and p521 have a computational complexity roughly half of the prime number bits. That is, 127 and 250 bits respectively. This would make p521 much stronger than Curve25519, though it executes significantly more slowly (roughly 10 times for a typical scalar multiplication). On the other hand, Curve25519 keys are only 43 base64 characters, whereas p521 keys use 87 of the same type, more than twice as long. It is desirable to use public keys that are as short as possible so users can handle them as they would handle a phone number. Here p521 still has the advantage because its security but, is it really necessary?

- The weakest point is actually the user-generated Keys that PassLok uses as private keys, after stretching. Experimental measurements of password entropy show that it is quite difficult for a user to remember a password having entropy larger than 60 bits. Key stretching cannot do much to increase the computational complexity of a weak key, which likely will be the subject of a dictionary attack or similar. For instance, 1024 ($=2^{10}$) rounds of SCRYPT stretching at r = 8, p = 1 take about the same computer time, using the latest JavaScript implementation, as a single Curve25519 scalar multiplication. Thus adding 1024 SCRYPT rounds would double processing time when obtaining a public key from a (weak) private key, equivalent to 1 bit of computational complexity. There is still a long distance to cover from the 60 bits of a user-memorized key to the 127 bits of complexity of a truly random Curve25519 key. This would mean increasing the processing time by a factor of $2^{127-60} = 1.47 \times 10^{20}$. If the unstretched computation took 1 ms, this would be equivalent to adding computations to take an additional 4.68 billion years. Clearly users are not going to put up with that.

- In other words, the user-supplied Key is the weak link, and no amount of key-stretching can fix this. There is no harm, therefore, in using Curve25519 rather than the p521 NIST curve in terms of real security. Since the Curve25519 public keys are half as long and everything processes much faster, we decided to switch to Curve25519, which meant also using the NaCl library rather than SJCL.

And now we discuss how PassLok uses all of these primitives. We begin with symmetric encryption and decryption, which is at the base of all encryption modes.

To encrypt a message with a symmetric key, PassLok normally goes through this process (except in Pad mode, described later, which does not use NaCl primitives):

1. Unless the message is meant to be sent by SMS (Short mode) or is a short string to be stored after it is encrypted, the plaintext is first compressed into a (usually shorter) base64 string by invoking the LZ-String JavaScript library. This library, also open-source like everything else included in PassLok, is chosen for its speed and ability to handle UTF16-encoded plaintext.
2. Key strength is evaluated by the WiseHash algorithm, developed by the PassLok team, and a variable number of SCRYPT iterations are applied to the key depending on the measured strength. The WiseHash algorithm, described in more detail later in this document, measures the bit entropy of the key based on the alphabet used, the number of characters (taking into account repetitions), and the presence of common words and their variants. It includes a

dictionary with the 10,000 most common English words and the 1,000 most common English passwords. Extension to other languages, although not currently implemented in PassLok, is easy to do without changing the basic code.

3. (SJCL version, prior to PassLok 2.2)
   When generating a key for use in AES, SCRYPT takes the key, plus an optional salt string, and generates a 264-bit stretched key from them (choosing 256-bit output was found to produce less than 256-bit output sometimes). If used to generate a private key, the final key length is instead 528 bits. SCRYPT is always used with parameters r = 8 and p = 1. The number of iterations, N (must be a power of 2), depends on the measured key entropy:

Entropy in bits	Iteration count N
< 10	4096
10 - 20	2048
20 - 40	1024
40 - 60	512
60 - 80	256
> 80	2

   (NaCl version, starting with PassLok 2.2)
   To generate a key for use in XSalsa20, SCRYPT takes the user-supplied key plus a salt string and generates a 256-bit stretched key by using parameters r = 8, p = 1, and a number of iterations calculated from the measured entropy of the key, in bits. The formula is: exp = max(1,ceil(20 - entropy/6)), so that the number of iterations is $N = 2^{exp}$. The maximum number of iterations, for known bad keys, is 2^{20}, far higher than in PassLok 2.1, and is more smoothly graded than in that version. Both formulas give 2 iterations for key entropy above 114 bits.
4. The stretched key and the compressed plaintext are then used as inputs to nacl.secretbox, which implements the XSalsa20 algorithm (SJCL.encrypt, implementing AES algorithm, prior to 2.2). Additional parameters are listed below. Notice that the key-stretching feature of the SJCL.encrypt function, based on the PBKDF2 algorithm, is reduced to the minimum allowed since there is already a full key-stretching step right before encryption.

	SJCL (up to PassLok 2.1)	NaCl (PassLok 2.2+)
Key length (bits)	256	256
Cipher mode	CCM (default)	n/a
Initialization vector (iv) or nonce	66 or 132 bits generated by the PRNG	9 or 15 random bytes, padded to 24 with zeros
Salt	Variable length, generated as described below	n/a
Authentication strength (bits)	64 (default)	n/a
Iterations of PBKDF2	101 (minimum allowed by SJCL)	n/a

The nonce or initialization vector is used to make sure that not two messages are encrypted with the same key. It is combined with the stretched key before this is used for encryption. Normally 132 bits (22 base64 characters) are used for the iv in SJCL, 120 in NaCl, unless it is a Short mode message,

where space is at a premium, or an encryption meant to store an item on the device. In those cases 66 bits (11 base64 characters) are used instead in SJCL, 72 bits in NaCl.

Even though it is no longer involved in the encryption process (it was with SJCL), an additional random-looking "padding" is computed in order to possibly contain a hidden message, for the main encryption modes. The padding string is computed this way:

   a. For a normal encryption, the PRNG produces a 256-bit random key and a 1,284-bit random "message". Then the random message is trimmed or padded to a pre-specified length (currently 59 characters, so a complete Lock can be transmitted), and then encrypted with the same parameters as a regular encryption and the same nonce as the plaintext encryption. The result is converted to base64.
   b. For a Decoy encryption, which includes a hidden message, the hidden message is trimmed or padded to the same pre-specified bit length as above and then encrypted using a special key for the hidden message and the same nonce as for the plaintext. The result, converted to base64, has the same length and looks as random as the result computed in (a) above, but it contains a message that can be revealed with the key.

5. After encryption, the resulting ciphertext, encoded as a uInt8 array made of 8-bit (0 to 255) byte values in decimal form is appended to an array that also contains a mode-identifying prefix, followed by the number of recipients (0 to 255), nonce, padding possibly containing a hidden message as described above, possibly a random public key (this is explained later) and, for multiple recipients, a series of user-identifying ID tags followed by an encrypted version of the message key for each recipient. Then the resulting array is encoded as a base64 string. The overall result may be bracketed by short human-readable tags whose purpose is to identify the encryption mode used.

To decrypt a message encrypted with a symmetric key, the process is this:

1. The tags are stripped and the base64-encoded data is converted into a UInt8 array consisting of byte values in decimal form. Then it is split into its constituents, in this order: mode-identifying code, number of recipients, nonce, padding, random public key (Anonymous mode only), user identifiers followed by their respective encrypted keys, ciphertext. The split is done at points dictated exclusively by location within the array, which is different for different modes. The modes are distinguished by the first character(s) after the initial tag, even in base64 form, according to this scheme:

Initial character	**Mode**
g	normal symmetric (used only in combination with Short mode)
d	Pad mode (symmetric)
h	Human mode (symmetric)
A	Anonymous (asymmetric) (if smallcase, Short mode)
S	Signed (asymmetric) (if smallcase, Short mode)
O	Read-once (asymmetric) (if smallcase, Short mode)
o	Read-once 2(asymmetric, Short mode)

p	Read-once 1(asymmetric, Short mode)
r	Read-once reset (asymmetric, Short mode)
k	directory item (symmetric)
l	Signature or sealed text (not encrypted)

2. Assuming that the key to the cipher text is already known (it will take additional steps, explained below, if this is not the case), the key is analyzed for strength and stretched with SCRYPT as described above for the encryption process. The stretched key should be the same as the stretched key obtained in the encryption process.
3. Then the decrypt function is called, using the same fixed parameters as for encryption, plus the nonce and cipher text extracted from the full encrypted message.
4. If the result is compressed plaintext (it can be easily detected if it is, for instance, an encoded file), the LZ-String decompression algorithm is called at this point. The result is finally displayed along with a confirmation message in a special area of the interface. If the decryption process fails (usually because the key entered is incorrect), a different message is displayed to warn the user. Errors are caught and sent to a function that displays a message telling the user what has failed, and possibly asks for specific data that might be missing.

Asymmetric encryption and decryption adds a few steps to this process, which are different depending on the asymmetric mode selected by the user and are explained later. In essence, the plaintext is still encrypted using the symmetric cipher and a symmetric key, but this symmetric key is obtained by combining the recipient's public key and a private key (the recipient's private key with a public key, for decryption) in different ways. But first we must discuss how the private key-public key pairs are generated in PassLok.

Public key generation

PassLok does not generate private keys. Instead, it accepts as a valid source for generating the private key whatever string the user wants to use as such. This is so that the user may remember the key and not feel tempted to write it down, which might easily lead to the key being compromised. Because many users tend to choose weak keys, PassLok applies a variable amount of SCRYPT key stretching, based on measured key strength as discussed earlier, before it actually uses it in any computation. This has two advantages over restricting the allowed key space:

1. If the user chooses a weak key, the program will run noticeably more slowly than with a strong key, because of the extra computations added by SCRYPT. This encourages users to choose strong keys that will be harder to guess.
2. All weak private keys remain as valid choices in the key space. A hacker wishing to guess a private key by generating public keys for entries in a special dictionary, for instance, will be forced to do a lot of expensive computations to test the weak keys, or risk missing those altogether.

NaCl uses private and public keys in a UInt8 array format, consisting of a fixed series of numbers from 0 to 255. The user-supplied key, which in PassLok is termed the "Key" (capitalized), is first subjected to a variable number of rounds of SCRYPT stretching, as described above, resulting in a 32-byte stretched

key. An optional salt value is added at each round in order to prevent the possibility of a powerful adversary generating a rainbow table containing the public keys matching all entries in a large dictionary as private keys. This salt value is requested from the user when the Key is first entered, and is stored locally (encrypted, more on this later) so the user does not have to enter it again, unlike the Key, which is never stored.

NaCl involves two kinds of private-public key pairs: on Curve25519 for performing Diffie-Hellman exchanges, and on the Ed25519 curve for digital signatures. PassLok uses the stretched user key as seed for a Ed25519 private key, which is made with this call in TweetNaCl:

KeySgn = nacl.sign.keyPair.fromSeed(stretchedUserKey).secretKey (64 bytes)

From this, the matching public key (called a "Lock" in PassLok) is made with this call:

Lock = nacl.sign.keyPair.fromSecretKey(KeySgn).publicKey (32 bytes)

And is immediately encoded in base64 for display or storage. Now, this is the key pair used for digital signatures. In order to do asymmetric encryption, a Diffie-Hellman key pair is needed. This pair derives from the first, by performing a change of variable from the Ed25519 curve to Curve25519. The calls involved are:

KeyDH = ed2curve.convertSecretKey(KeySgn) for the secret key (32 bytes)

LockDH = ed2curve.convertPublicKey(Lock) for the public key (32 bytes)

The functions ed2curve.etc, although not included in NaCl itself, were added by the author of the JavaScript implementation of NaCl and are available from the same GitHub site.

At the end of this process, the public key can be displayed. This is what a typical PassLok public key, also known as a Lock, looks like:

PL21ezLok=1iw+0_y5xyh_66nby_u12x1_hmdw8_iioou_6yhud_a8/i9_jd4fj_fvv6i_swkrn_u773t_jb7yr_+d9nn_/b4h6_880py_vtf4L_o4zwr_6207u_v/bdd=354ad_7836e_52c1a_2cae9=PL21ezLok

Or even shorter:

PL22lok=FWDpcOXA0AuW0SC/JLc2NMngivPM9zuDuY923UMqhkc=PL22lok

PL24ezLok==327k7-e5x30-ntbfk-konLe-tazzy-5Ldxc-7tebe-xLp04-ysnfv-50ipg==PL24ezLok

The reader hopefully can appreciate how much more compact this is, even with the error-correction code added at the end in the first example, than the typical PGP public key.

When a Lock is to be used for verifying a signature, the only conversion needed consists in decoding the base64 characters back to a Uint8 byte array. If it is going to serve as public key for a Diffie-Hellman exchange, it is first passed through the ed2curve.convertPublicKey function in order to obtain its

Curve25519 equivalent. This step is skipped for ephemeral keys (described later), which are generated from the start on Curve25519 rather than on Ed25519.

Asymmetric encryption and decryption

PassLok implements three main modes of asymmetric encryption, which are user-selectable via the Options tab:

1. Anonymous. The message is encrypted so the recipient can decrypt it with his/her private key. The recipient cannot know who encrypted it, from the ciphertext. It does not mean "anonymous" in the special sense of being untraceable over a network.
2. Signed. The recipient must use his/her private key and the sender's public key, or otherwise decryption fails. This is equivalent to applying a digital signature to the plaintext before encryption, hence the name.
3. Read-once: The private key/public key pair changes for each message exchanged between a given sender-recipient pair, so that someone intercepting the encrypted messages would be unable to decrypt it after the exchange is finished, even if he/she/it manages to obtain the permanent private keys of the participants. The messages also enjoy deniability, since it becomes impossible to detect who the sender or the recipient is, even if permanent private keys are compromised later on.

These variants are often available, alone or in combination, within those main modes:

- Short: The encrypted message is less than 160 characters long, so it fits within a single SMS message.
- Decoy: There is a hidden message in addition to the main message, encrypted under a separate key. Its presence is undetectable without possessing the appropriate decryption key.
- Chat invitation: This is a specially encrypted message, which will automatically open a secure real-time chat session as soon as it is decrypted. The chat takes place within a sandboxed frame, so that the main PassLok code and its data remain unaffected by anything happening in the chat.

Except for Short messages, which don't have enough space for this, all modes admit multiple recipients whose identities remain unknown to one another. We proceed now to explain how the different modes of asymmetric encryption are put together, in this order: Signed, Anonymous, Read-once. The default choice without user intervention is Anonymous mode.

Beginning with version 2.3, users can select an Email interface mode, where only the functions used in PassLok for Email (a Chrome extension that integrates with popular email services) are displayed. In this case, Anonymous encryption is not available and the default is Signed mode.

Signed encryption/decryption

Signed mode encryption proceeds this way:

1. The message is compressed with LZ-String (unless it is detected to be an encoded file, which typically won't compress well) and encrypted with XSalsa20, using a 15-byte random nonce (9-byte for Short messages), a 100-byte padding, obtained as described in the section on symmetric encryption, and a random 256-bit message key. If Decoy mode is engaged, the padding results from encrypting the hidden message, otherwise it consists of random bytes. The resulting cipher text is stored in memory.
2. For each recipient, the following is done:
    a. The sender's stretched private key was obtained by stretching the user Key, followed by an ed2curve.convertSecretKey call when PassLok first loaded, and is retained in memory.
    b. The recipient's Curve25519 public key is obtained from his/her Lock by means of a ed2curve.convertPublicKey call, preceded by decoding from base64.
    c. Both keys are combined into a DH shared secret by means of this operation, but the result is not stored (unlike when using SJCL) because the operation is fast and can be repeated without much delay to the whole process:
    sharedSecret = nacl.box.before(publicKey,privateKey)
    d. Take the recipient's Lock and encrypt it with XSalsa20, using the main message's nonce, and the recently calculated shared secret as encryption key. The first 8 bytes of the resulting ciphertext are this recipient's "ID tag."
    e. Take the main message key and encrypt it with XSalsa20, using the same parameters as in the step above and the same shared secret as encryption key. Put the resulting cipher text in memory.
3. Then PassLok concatenates the bytes obtained above in this way: initial tag (optional), single byte of decimal value 72 (begins with "S" when encoded as base64), single byte with the number of recipients (1 to 255), nonce or iv, padding, then for each recipient the iD tag followed by the message key encrypted with the recipient's shared secret, and then finally the message cipher text. Then the result is encoded as base64 characters.
4. The item is then bracketed by readable tags so users can identify the kind of item this is. If Email mode is engaged, the sender's ezLok (public key encoded in base36) is prepended to the encrypted message, right before the type indicator, and longer, more descriptive tags are used instead of the standard ones.

Decryption is done this way:

1. Strip the initial and final tags and retrieve the initial ezLok (public key) if present, then decode the string from base64 back to UInt8 array and split it into its components. The components are: mode indicator, number of recipients, nonce, padding, then a series of ID tags and encrypted message keys in alternation, and finally the cipher text. Mode is actually detected before the decoding from base64, since the first character in Signed mode should be "S". There is no need for the user to set anything so the correct processing mode is selected.
2. The user now needs to find the instance of the message key that has been encrypted for him/her to decrypt. This item follows immediately after an ID tag that is generated this way:

a. The user's DH private key should have been in memory from the moment PassLok loaded. If not, generate it again from the user's Key and salt string, through the key-stretching process described earlier, followed by conversion to Curve25519.
b. Take the user's own Lock, which should also exist in memory. If not, PassLok generates it again from the user's signing private key.
c. Take the sender's Lock, which is supplied in a special input field. If this is unavailable, PassLok stops and directs the user to do so. Then make its Curve25519 counterpart with ed2curve.convertPublicKey.
d. Combine the DH private key with the sender's DH public key by:
sharedSecret = nacl.box.before(publicKey,privateKey)
Because of the commutative property of scalar multiplication over elliptic curve fields, the result should be identical to the result obtained by the sender in step 2c above.
e. This is the shared secret held in common by the sender and the user. Now use it as key to encrypt the user's public key with XSalsa20, with the same nonce as the main message, and the usual additional parameters. The first 8 bytes of the resulting cipher text constitute the "ID tag" for this message.

3. Search for the ID tag among the components of the message. If no exact match is found, display a warning saying "there is no message for you" and stop. If there is a match, the component immediately following it is the encrypted message key.
4. Take the encrypted message key and decrypt it with XSalsa20, using the shared secret as key, and the nonce already recorded. The result, if successful, is the message key. If unsuccessful, display a warning.
5. Take the cipher text and decrypt it with XSalsa20, using the decrypted message key, and the same nonce. If successful, the result is the compressed plaintext. If unsuccessful, display a warning. The final step is to decompress the plaintext with LZ-String (unless it is detected to be an encoded file).

The overall process is not very different from the way other programs, such as PGP, handle multiple asymmetric encryption: the plaintext is symmetric-encrypted with a random message key, and then the message key is asymmetric-encrypted for each recipient, but there are some traits proper to PassLok:

1. Recipients can identify only the message keys that have been encrypted for each of them, so that identities of the other recipients remain unknown. It is tempting to use a part of their unencrypted public keys as ID tags, but then the identities of the recipients would be revealed to outsiders. This is why the ID tags are based on encrypted public keys, and the encryption keys for this are the same that would decrypt the corresponding message key. Doing this does not add much computational effort since the most expensive step is the elliptic curve multiplication needed for the Diffie-Hellman exchange, and this has to be performed regardless. The ID tag for a given recipient is different in each message since it depends on the message's nonce value.
2. The sender's identity cannot be obtained from the message, either, until some parts of the message are successfully decrypted. The only way anybody can identify the sender is by successfully making at least one of the ID tags included with the message, which implies making

a shared secret by combining the sender's public key and the recipient's private key. No one who is not in possession of one of these private keys can make an ID tag that will be found in the complete message string. Those who do get a first confirmation of the sender's identity when the ID tag is found, and again when the message key, which is encrypted with the same shared secret mentioned above, is successfully decrypted.

3. Unlike in other cryptosystems, where "signed" messages involve a digital signature followed by Anonymous asymmetric encryption, so that signature verification is carried out after decryption, in PassLok both steps are combined into one, which speeds up the process substantially. The sender authentication instrument is his/her public key, just as in signature verification, but it is used in the decryption itself. If the sender's public key is incorrect, the shared secret obtained in step 2d above will be different from the one used by the sender, and the resulting ID tag will also be different, leading to no match when the ID tag is searched.

Anonymous encryption/decryption

In PassLok, the term "Anonymous" is used in the general sense of not knowing the identity of the originator of a message, rather than the restricted sense used in the digital world to mean that one cannot be tracked over a network. A reader well versed in cryptography might prefer to refer to this mode as "deniable" rather than Anonymous, but we'll keep using the latter term for the sake of consistency with the interface. The process followed for this mode is very similar to that of the Signed mode, but there is one fundamental difference. In Anonymous mode, the sender uses an ephemeral, random private key, rather than his/her permanent private key. From this random private key, an ephemeral public key is derived with the command nacl.box.keyPair.fromSecretKey(privateKey).publicKey. Since the random key already has a 256-bit entropy, SCRYPT stretching is not used. The ephemeral public key is placed in the outgoing message immediately after the padding, before the pairs of ID tags and encrypted message keys.

Upon decryption, once Anonymous mode is detected by the presence of the "A" character immediately behind the initial tag, PassLok knows that an ephemeral public key follows the padding, and extracts the components accordingly. The shared secret is calculated from the user's permanent private key, which was calculated when PassLok loaded, and the ephemeral public key, rather than the sender's permanent public key. There is no point in storing the resulting shared secret since it will be different next time. Other than this, the process is identical to the one described for Signed mode.

PassLok's Anonymous asymmetric encryption bears some similarities to the ElGamal encryption algorithm. In both cases a random value is used as private key and then the Diffie-Hellman public key is derived from it and sent along with the encrypted message. But in ElGamal encryption the plaintext (or, as in PassLok, the symmetric message key) is operated on by the shared secret using large-integer multiplication or elliptic curve addition, while in PassLok the shared secret is used to encrypt it with the symmetric cipher.

Observe that the sender's permanent private and public keys are never involved in the process. There is no way to know the sender's identity from the encrypted message, and this is why the mode is termed

"Anonymous." When the message is decrypted, the recipient does not have to supply the sender's "public key," since this is already included as a component of the full encrypted message. ID tags and encrypted message keys are encrypted by the shared secret resulting from the DH combination of the recipient's permanent public key and the ephemeral private key. On the receiving end, they are encrypted with the shared secret resulting from combining the recipient's private key and the ephemeral public key that accompanies the message, which is the same for all recipients. As in Signed mode, the resulting shared secret is the same computed either way.

Even though the ephemeral private key used is the same for all recipients, there is no interference between them since the shared secrets that actually encrypt the different instances of the message key and are used to make the ID tags are also based on their respective permanent public keys, which presumably are all different, so that the respective shared secrets are also all different.

Read-once encryption and decryption, mode 1

Read-once mode 1 is one step beyond Anonymous mode. Let us involve two correspondents, Alice and Bob, in order to understand better how this works. Alice has just sent an Anonymous encrypted message to Bob, who wishes to reply. Rather than replying using Alice's permanent public key, Bob uses the ephemeral public key that Alice sent along with her last encrypted message. Alice will be able to decrypt it if she still has the ephemeral private key that she generated for that last message. For her reply to Bob's new message, Alice uses the ephemeral public key that Bob sent along with his message, and generates a new random private key and its matching public key, which she again sends along with her new reply. Thus a sort of ping-pong game takes place between Alice and Bob, where each new message encrypted implies generating a new ephemeral private key and its matching public key, as in Anonymous mode, but which will be used again for the reply, unlike in Anonymous mode.

Every time one of them receives a message from the other, the message includes a new public key, which will be used for the reply instead of the previous one, which is overwritten. Likewise, every time a reply is made, a new private key is generated and the previous one overwritten. Since subsequent messages are encrypted with shared secrets resulting from combining each time a different private key or public key, once either of them is overwritten the message can no longer be decrypted. Therefore, perfect forward secrecy of previous messages is achieved as more messages travel back and forth between the correspondents, since they were encrypted with ephemeral keys that have been overwritten on both ends of the conversation.

Since PassLok is meant to supplement asynchronous communications, Read-once mode requires storage. Alice wants to be able to decrypt Bob's reply, which will be encrypted using the ephemeral public key she sent out with her previous message, so she needs to store the matching private key. She also needs to store the ephemeral public key that Bob sent, in order to reply to him. In PassLok, a local directory is set up where data pertaining to each particular recipient (or sender) is stored. It takes the form of a JavaScript array. In the case of the array pertaining to Bob, stored in Alice's computer, the contents are the following, where those marked as "encrypted" on the table are encrypted with Alice's permanent Key (SCRYPT-stretched with the user name as salt) before they are stored:

Index	contents
0	Bob's public key (unencrypted by default)
2	Ephemeral private key last used to encrypt for Bob (encrypted)
3	Ephemeral public key from Bob's last message (encrypted)
4	Boolean flag indicating whose turn it is to encrypt (unencrypted)

In order to encrypt a Read-once message for Bob, or decrypt a Read-once message from Bob, Alice must point PassLok to this array so the appropriate strings can be read or stored. It follows that users must select the appropriate correspondent both for encryption and for decryption, as in Signed mode. This forces a design choice. Recipients cannot be sure of the identity of a Read-once message sender since the ephemeral key pair involved in making it is randomly chosen, as in Anonymous mode, so it would seem that users should deal with Read-once messages very much like they would with Anonymous messages. But the fact that they must identify the sender so the new public key can be stored in its proper slot makes it feel to the user like some sort of authentication is taking place. In fact, the identity of the sender is not being verified.

Further, consider this: if the same ephemeral key pair is used for all the recipients of a given message, it would be possible for any of them (or a third party who gains access to the public key sent along with the message) to impersonate any of the other recipients when replying to the sender. He/she/it only needs to come up with a new ephemeral key pair, use the private key in combination with the previous sender's public key to encrypt the message, and send the new public key along with the message.

Users naturally would be confused about identifying senders whose identity in fact cannot be verified, and might end up placing trust where it shouldn't be placed. Therefore, we decided to implement Read-once mode in a way that senders actually are subject to authentication. We did this by using a *different ephemeral key pair for each recipient* of a given Read-once message. Each ephemeral private key is locally stored, after encoding as base64, at index 1 of the array assigned to its recipient, and the matching public key is encrypted in the same way as ID tags and message key, and added to the material following that particular recipient's ID tag in the full encrypted message.

This ID tag, like the items immediately following it, cannot be encrypted with the shared secret resulting from combining the recipient's previously sent public key and the new ephemeral private key, since the recipient would need the matching public key to decrypt that same public key out of the message. At this point we have several options for encrypting the ID tag and the new ephemeral public key. The simplest one is to use the permanent shared secret, resulting from the combination of the sender's permanent private key and the recipient's permanent public key is used (the actual message key would still be encrypted with the combination of the recipient's stored ephemeral public key and the sender's new ephemeral private key for this particular recipient). But this choice would make the sender and recipient identifiable from the ID tag if their permanent secret keys are compromised in the future (in technical words, the message would lack "deniability").

A better choice, which does not require additional storage, is to encrypt the ID tag and new ephemeral public key with the shared secret resulting from combining the sender's permanent private key and the recipient's most recent ephemeral public key, if there is one (otherwise the recipient's permanent public key is used). Then the recipient can compute the same shared key from his/her most recent ephemeral private key if there is one (otherwise, the permanent private key) and the sender's permanent public key. This scheme still does not ensure deniability of the messages if all of them from the first one are available to an attacker, since then he/she/it would be able to obtain all the ephemeral public keys in succession and thus the ID tags could be reproduced, but if just one of the messages is not recorded then the whole chain from that spot onwards becomes deniable.

When decrypting the message, the recipient first computes the ID tag, which is very much like the ID tag computed in Signed mode. This provides sender authentication since the ID tag can only be made successfully by someone possessing the sender's permanent private key (or the recipient's permanent private key, which would be a much more serious problem by itself). Once the ID tag is found in the full encrypted message, the encrypted ephemeral public key is found, decrypted with the same key used to make the ID tag, and stored at index 2 of the array assigned to that sender (actually, storage happens at the end of the process, to avoid corrupting the stored keys with material from messages that fail to decrypt). Then the ephemeral shared secret is made by combining this public key and the private key for this sender, which was stored when the recipient last encrypted something for this sender, and the message key is decrypted. Finally, the plaintext is obtained by decrypting the cipher text with the message key. All symmetric encryptions and decryptions are performed with the same nonce and the same optional parameters listed above, which represents an insignificant risk since the plaintext encrypted is random-like and thus conventional attacks on re-used nonces would fail.

Whenever a required ephemeral key (private or public) is not found in storage, the corresponding permanent key is used instead. For instance, If Alice is initiating the first Read-once message to Bob, with no previous history of Read-once messages between them, she will use Bob's permanent DH public key, rather than an ephemeral key that doesn't exist, and an ephemeral DH private key that she generates just then. When Bob replies, he will use a newly generated private key and the public key that Alice sent along with her message, thus populating both ephemeral strings on his side. Alice will fill the second ephemeral slot as soon as she decrypts Bob's message, and from then on they will use exclusively ephemeral keys, changed at every exchange back and forth, to communicate with each other.

Here is a step by step diagram of a series of Read-once exchanges between Alice and Bob. Small case single letters are private keys, capitals are matching public keys. Both are permanent if unnumbered, ephemeral if numbered. Parentheses denote symmetric encryption using the Diffie-Hellman combination of the private and public key inside the parentheses as key. All exchanges are recorded after the message is decrypted by the recipient. The permanent private keys, a and b, are not really stored, but rather input anew for each message. Some slots are not immediately filled; when this happens, the word "null" is used to represent the empty slot.

Exchanges	Alice storage	Sent by Alice	Bob storage	Sent by Bob
1st ms. from Alice	a,B,a1,*null*	A1(a,B),ms1(a1,B)	b,A,*null*,A1	
1st ms. from Bob	a,B,a1,B1		b,A,b1,A1	B1(b,A1),ms2(b1,A1)
2nd ms. from Alice	a,B,a2,B1	A2(a,B1),ms3(a2,B1)	b,A,b1,A2	
2nd ms. from Bob	a,B,a2,B2		b,A,b2,A2	B2(b,A2),ms4(b2,A2)

The first message from Alice can only be decrypted as long as the pairs (a1,B) or (b,A1) can be collected. After the second message from Alice is decrypted, the ephemeral keys a1 and A1 have been overwritten on both ends, and so the first message can only be decrypted if the secret key b is compromised, because then A1 can be decrypted. The second message is encrypted by ephemeral keys only, and thus when those are overwritten, after the fourth message is sent, it can no longer be decrypted even if the permanent private keys, a and b, are compromised and access is gained to the correspondents' machines, where the ephemeral keys are stored symmetric-encrypted by their respective permanent private keys. At the end of the series, both Alice and Bob reset their storage so that the current ephemeral keys are deleted, and then no message in the series can be decrypted even if their private keys are compromised. Even more, in order to be able to associate the messages with correspondents whose permanent private keys have been revealed, all of the messages exchanged need to be obtained.

It follows that correspondents need to take special care with the first message, which initiates the exchange, since this one does not possess forward secrecy or deniability. After this first message, however, all messages have both properties, and can be posted in public without an attacker ever being able to decrypt them or even link them to the correspondents.

Read-once conversations can also be established between correspondents who share a common symmetric key, rather than private/public key sets. In this case, a public key is made from that shared secret, taken as private key, when the recipient's permanent public key is needed to encrypt the first message of the conversation. ID tags and ephemeral public keys are encrypted directly with the shared symmetric key, in all exchanges. Ephemeral keys are stored exactly as for all other correspondents.

Read-once mode 2

Observe that, every time one of the correspondents decrypts a new message, there are two ephemeral public keys present in memory (the previous one, and the one that comes with the new message) before the old key is overwritten. This offers the possibility of a slight variation: encrypt the message not with the new ephemeral private key, but *with the old one, if there is one*. Then the recipient will use the old ephemeral public key, if there is one. The exchange now looks this way (differences with Read-once mode 1 are highlighted in boldface):

Exchanges	Alice storage	Sent by Alice	Bob storage	Sent by Bob
1st ms. from Alice	a,B,a1, *null*	A1(a,B),ms1(a1,B)	b,A,*null*,A1	
1st ms. from Bob	a,B,a1,B1		b,A,b1,A1	B1(b,A1),ms2(b1,A1)
2nd ms. from Alice *	a,B,a2,B1	A2(a,B1),ms3(**a1**,B1)	b,A,b1,A2	
2nd ms. from Bob	a,B,a2,B2		b,A,b2,A2	B2(b,A2),ms4(**b1**,A2)

The exchange begins as in Read-once mode 1 but by the third message, marked with an asterisk, Read-once mode 2 is in full force. This message is encrypted with the first of Alice's ephemeral key pairs, which is overwritten, on both sides, as soon as that message is acted on: immediately after encryption, on Alice's side, and immediately after decryption, on Bob's side. The result is that Alice cannot decrypt the message she has just encrypted, and Bob can only decrypt it once. This is akin to the message "self-destructing" after it is read. There is no need to reset the stored keys when the conversation is over, since none of the messages (except the first one) can be decrypted even if the permanent private keys are compromised. The second and third messages exchanges were encrypted with the same shared key, but since this was based on ephemeral keys, it is not possible to decrypt them after the third message is decrypted. Deniability is achieved starting with the third message, since the key used for encrypting new ephemeral public keys and ID tags is the same after that point, and it can no longer be reproduced.

Since the stored values are the same in Read-once mode 1 and Read-once mode 2, they can be used interchangeably within an ongoing conversation. PassLok detects the mode used in a given encrypted item from the first character after the initial tag on short-mode messages ("p" for mode 1 and "o" for mode 2, and there is still a special reset mode, which begins with "r"), or from a single byte following the encrypted ephemeral public key in regular messages (see it at the end of this paragraph) and performs the decryption accordingly. Other than this, the structure of the encrypted item is the same: initial tag, mode indicator, number of recipients unless it is Short mode, 15-byte random nonce (9 bytes in Short mode), 75-byte padding (which may be random or contain a hidden message), then for each recipient the ID tag plus single-byte type indicator plus encrypted message key plus encrypted ephemeral public key, ciphertext (symmetric-encrypted by message key), final tag.

Read-once mode 2 is less resistant to changes in the back-and-forth order of the exchange than mode 1. In mode 1 it is possible to decrypt a message many times before it is replied to; in mode 2, the decryption fails the second time this is attempted. In mode 1 the sender can re-encrypt a new message for the same recipient immediately after encrypting another; so long as the first encrypted message is not sent, the exchange remains in sync. Even if several messages are encrypted and sent without waiting for a reply, the recipient can read them all and the exchange remains in sync so long as the last message decrypted is the last message encrypted. In mode 2, if a sender changes his/her mind about the message he/she just encrypted, encrypting a new message would throw the exchange out of sync so that no subsequent messages would be successfully decrypted on the other end. To prevent this, PassLok keeps track of whose turn it is to encrypt by means of a Boolean flag that is stored along with the ephemeral keys.

PassLok defaults to Read-once mode 2, but attempting to encrypt out of turn causes PassLok to use mode 1 instead of mode 2. After a reply from the recipient is successfully decrypted, mode 2 encryption for that recipient is allowed once again. Also, the ephemeral private key will not change so long as the sender is encrypting in mode 1, in order to prevent the problem discussed in next paragraph.

Let's say that the ephemeral private key is changed for every outgoing message, whether in mode 1 or 2, and let's further imagine that Alice sends several Read-once messages without waiting for a reply from Bob. These would be encrypted in mode 1 after the first, which would be in mode 2. Bob will be

able to decrypt the mode 1 messages without a problem because the ephemeral public key used is contained in each message, but if he decrypts them in reverse order the public key stored won't be the last one. If he now replies to Alice in Read-once mode (mode 2, in this case), the public key he uses (not the last one generated) won't match the private key Alice has stored on her side (the last one), and the reply will fail to decrypt.

Using the same ephemeral private key until a message is received solves this problem, at a small cost to forward secrecy, which won't be acquired for any of the messages using the same private key until a reply is made, at which moment all of them become forward-secret at the same time. There are other scenarios that will cause the conversation to go out of sync. In order to put the conversation back in sync, the correspondents must clear the ephemeral data stored and restart the process. PassLok makes this easier by adding a "reset" flag to encrypted messages. Let's say Alice clears the data pertaining to her exchange with Bob, which also sets the turn flag to "reset" rather than "lock" or "unlock". Next time Alice sends a Read-once message to Bob, a reset indicator accompanies the encrypted message key, rather than a mode 1 or 2 indicator. When Bob decrypts this message, PassLok recognizes that a reset is requested and clears the ephemeral data pertaining to Alice before it proceeds with the rest of the decryption process, which now continues as if this was the first time Read-once mode has been used between them.

PassLok forward secrecy vs. OTR and Signal

When the Read-once mode in PassLok was designed, the author was unaware that the Off-the-Record protocol (OTR), and also the Signal protocol, use a very similar trick in order to achieve forward secrecy. OTR is closest to Read-once mode 2, since the shared key used to encrypt a given message is based on the previously received public key, rather than on the one accompanying the message. The way OTR keeps the conversation in sync, however, is different. If Alice wants to encrypt a new message for Bob before receiving Bob's reply to her previous message, OTR reuses the ephemeral private key that was used in that previous message, and sends out the same ephemeral public key along with it, just like PassLok, but it won't use them to encrypt the new message as in PassLok's mode 1. Instead, OTR will keep encrypting the messages with the previous ephemeral key (as in PassLok's mode 2), which won't be erased from storage for an additional cycle.

This requires *two* sets of ephemeral keys to be stored at any given time: the preferred one, and the one to be used out of sync, which is exactly one generation older. As a consequence, encrypted messages do not become absolutely un-decryptable until a reply has been encrypted and received, since the older ephemeral keys linger in storage for an extra cycle. In other words, they might as well have been encrypted using the most recent public key, as in PassLok's Read-once mode, as far as forward secrecy goes.

Perhaps one of the reasons why OTR does not use the most recent public key is that others would be able to impersonate the recipient if they also got that public key, as was mentioned earlier. While PassLok transmits ephemeral public keys in encrypted form, OTR transmits them in plaintext, so that using the most recent public key would be insecure.

PassLok can somehow get around the sync problem because it is not specifically designed for instant messaging like OTR. A PassLok user can see what kind of forward secrecy a given message has, and act accordingly, whereas an OTR user most likely would not get a visual indication since everything is handled automatically by the instant messaging program. This being the case, OTR cannot afford to use several types of forward secrecy without compromising this property for the whole conversation. On the other hand, a PassLok user that wishes to send two Read-once messages in a row switches automatically to mode 1, instead of the default mode 2, until a reply is received. Mode 1 messages can safely be received out of sequence, and in this case the ephemeral public key that is stored last (presumably from the last message received) is the one involved in the reply; the other correspondent will be able to decrypt this reply if the last message he/she sent was also the last received. Resetting the conversation is always a last resort.

OTR involves a complex system for mutual authentication and deniability. Signatures are used for the initial exchanges, and there is a special set of steps for revealing ephemeral authenticating data after the conversation is over, so that anyone can forge an authentication at that point. PassLok takes a simpler approach without signatures, MACs, or special authenticating keys involved at any point. Correspondents initiating a PassLok conversation in Read-once or Read-once mode authenticate each other as their permanent private keys are involved in the initial exchanges. Once a conversation is in course, authentication is assumed in PassLok, since only those able to retrieve the next public key, which comes in encrypted, will be able to continue the conversation. OTR, on the other hand, sees the need to keep authenticating the correspondents at each message by means of a new MAC using a hash of the current shared key as MAC key (the first message is authenticated by means of signatures based on the permanent private keys, so neither is it deniable nor does it possess forward secrecy). It is likely possible to communicate with several recipients at once while doing this, but there is no doubt that it is going to be more complex than the way PassLok does it.

The Signal protocol, which is used by WhatsApp and other messaging apps, started out as a variation on OTR, but in its most recent versions it has evolved a “double-ratchet” method to keep the conversation in sync. In essence, a new ephemeral private key is generated every time a message is received from a certain sender (the old one is overwritten immediately), and the shared secret is not used directly for encryption, but rather as the starting point of a “ratchet” (actually, a series of successive hashes, to be used as message keys) for messages sent or received until the next ephemeral public key is received. Every message key within a ratchet has a consecutive serial number, which is transmitted in plaintext along with the message, and every message key is stored until a new ratchet is generated. This allows each message to use a different symmetric key, even if several are encrypted by the same sender in quick succession. Since the shared secret can be generated on both ends, all the keys in the ratchet can be produced on both ends so long as its number is known, and all messages can be decrypted even if they arrive out of order.

Signal uses this trick, which has been considered secure enough by the experts that have analyzed it, so no message is lost, but it does entail the local storage of additional secrets, which cannot be erased until the corresponding messages have been decrypted on the other end. This creates a liability in the event of one of the correspondents’ machines being compromised. Local storage is presumed encrypted, but it

would be better if secrets were not stored at all. PassLok does re-use the message key until a reply is received bringing in new key material, but compromising this (ephemeral) key only affects the messages encrypted with it. A similar compromise in Signal would preserve the security of the messages on the same ratchet encrypted before the compromise, but those following it would be compromised as well.

The differences between the way PassLok on one hand, and OTR and Signal on the other hand, handle forward secrecy can be attributed to the latter two being instant messaging apps whereas PassLok is more focused on asynchronous conversations. Losing a message, or being unable to decrypt it, which amounts to the same, is more unpleasant in a synchronous conversation than on a chain of emails, for instance, and so OTR and Signal go to extra pains to make sure that no message is lost, incurring the liability associated with storing extra secrets. But in asynchronous communications any storage is liable to turn into permanent storage, so the emphasis rather is on storing as little as possible that might later be compromised, even if doing so might cause some messages to be lost. In that event, the recipient can ask for the lost information to be re-sent if it seems necessary.

Other cryptographic functions

Short mode

When Short mode is selected on the Options tab, a simpler algorithm is chosen. In order for the final result to fit within the 160-character limit of an SMS message, only single recipients can be selected in Short mode. This eliminates the need for ID tags and individually-encrypted keys, resulting in additional space savings. Nonces comprise only 9 bytes.

Since there is only one recipient, the plaintext is symmetric-encrypted directly by the Diffie-Hellman combination of the sender's DH private key (or an ephemeral key, for Anonymous mode) and the recipient's permanent DH public key. If symmetric encryption is used, the shared symmetric key is used directly to encrypt the plaintext, after appropriate stretching, and a special single-byte mode indicator (which causes the first character of the base64-encoded output to be "g") is prepended to the output. The plaintext itself is of limited length, and a message warns the user of the number of characters remaining as the message is being typed. Signed and symmetric encryption allow for 94-character messages, Anonymous for only 62 characters, since the ephemeral public key takes 32 bytes that otherwise would be used by plaintext. Read-once and Read-once modes can accommodate only 46-character messages because the encrypted ephemeral keys have a 16-byte MAC attached to them. Characters beyond the limit are truncated and the user is warned.

The character limit applies to ASCII encoding. Plaintext characters outside the ASCII subset, even if correctly handled by SJCL, lead to longer ciphertext, thus creating encrypted messages that would not fit within the 160-character limit. In order to prevent this, the plaintext is subjected to an encodeURI operation, which replaces non-ASCII characters with ASCII strings, prior to encryption. Since this operation encodes spaces as "%20" strings, those are replaced back by spaces before encryption in order to maximize character utilization. No LZ compression is used for short messages, since this increases the length rather than decrease it when the message is so short. Upon decryption, the resulting plaintext is subjected to a decodeURI operation in order to recover the original plaintext.

Other than the convenience of making SMS-length encrypted messages, Short mode is useful for development. The algorithms used are the same as for the regular modes, except without the need to make ID tags etc., which helps for debugging the regular encryption modes.

Digital signatures

PassLok has the ability to create digital signatures for a plaintext using the user's private key, and to verify an existing signature by means of the signer's public key. NaCl performs the Schnorr signature scheme on the Ed25519 elliptic curve, which forms the basis of PassLok signatures. Essentially, it goes like this for signing a message with a private key x:

1. Choose a 256-bit random number k

2. Compute r = k**g, where g is the base point of curve Ed25519 and ** denotes scalar multiplication of a number k and a point g on the curve.

3. Take the SHA512 hash of the text to be signed, concatenated with r, which will be called e.

4. Compute s = k – x*e, where - and * are operations modulo p, where p is the prime number of the Ed25517 field = 2^{255} - 19.

5. The signature is the pair (s,e), usually output in base64 encoding.

To verify the signature, the verifier does the following with the signature (s,e) of a given text, plus the verifying public key y:

1. Compute r_v = s**g + e**y, where the new operation + is point addition on the Ed25519 curve.

2. Take the SHA512 hash of the text concatenated with r_v, and call this e_v.

3. If e_v = e, the signature is verified, otherwise verification has failed.

All of this works because y = x**g, and therefore r_v = s**g + e**y = s**g + e**(x**g) = (s+e*x)**g. Then since s = k – x*e, r_v = (k – x*e + e*x)**g = k**g = r. And therefore the hashes e and e_v are also identical.

The random value k is replaced in NaCl with the hash of the text concatenated with the private key, which is just as unique to the signature as a true random number, thus eliminating the risk of a faulty PRNG.

In order to maintain single-box input, PassLok appends the signature to the text to be signed, leaving a blank line between them, rather than putting the signature in a different element. When verifying the signature, PassLok first separates the signature from the original text, and then performs the operations summarized above. Care is taken to eliminate spurious line feeds that might have been introduced into the signed message, usually as a consequence of default formatting of the intervening containers. Since the formatting of the signed text might be important to its authenticity, PassLok limits itself to replacing
 and tags with newline characters (the matching tags are filtered out) leaving all other HTML formatting tags intact, before taking the hash.

Beginning with version 2.2.0, PassLok added a second signing mode, called a “Seal”, which can be selected from the Options tab instead of the default “Attached” signature. In this mode, which is built into NaCl (it doesn’t exist in SJCL), the signed text is wrapped by the signature so that the result is a random-looking item containing the text itself. When a sealed item is verified, the original text is recovered at the same time as the signature is verified. This mode produces longer output, but has the advantage that the text cannot be undetectably modified through formatting, spacing, etc. before verification, thus causing signature verification to fail. This mode became the only one available beginning with version 2.3, so that all sealed messages actually contain the plaintext. Starting with version 2.4, a series of padding bytes were added to sealed messages, which might contain a message hidden in Decoy mode, as described below.

Decoy mode

We have mentioned this optional mode earlier, so in this section we will only present some additional information. Decoy mode supplements most other encryption modes by encrypting a hidden message, which is then placed as the "padding" string of several PassLok items. Short mode messages don't support Decoy mode for lack of space.

The purpose of Decoy mode, which gives it its name, is to provide a hidden channel to combat the "rubberhose attack." In this scenario, the recipient of an encrypted message is forced to disclose his/her private key (possibly by repeated application of a rubber hose or similar instrument to a part of his/her anatomy), so the message can be decrypted. Decoy mode makes it possible to exchange sensitive information by means of the hidden messages while the main encrypted messages are "decoys" containing harmless information. This gives the recipient "plausible deniability." That is, he/she can claim that there is no additional hidden information since that hidden information, even if present, is undetectable. In the absence of other indications to the contrary, a rational enemy would conclude that the only information contained in the encrypted message is what can be obtained by decrypting it in the normal way. Another use scenario is as a "canary" which, when present and containing the correct information, will assure the recipient that the sender did not act under duress.

Hidden messages are limited to 75 ASCII characters at most, so that a complete Lock can be transmitted this way. As in Short messages, the plaintext is subjected to a modified encodeURI operation before encryption, to be reversed by decodeURI after decryption. PassLok provides feedback to the user as the hidden message is entered so its length does not exceed the limit. Characters beyond the limit are truncated and the user is warned of this fact.

Encryption of the hidden message can be of two types: symmetric and asymmetric. PassLok decides which mode to use from the length of the special key supplied along with the message. If the length of its base64 part is exactly 43 characters, PassLok interprets it as a public key and uses asymmetric encryption, otherwise it uses symmetric encryption. Symmetric keys are stretched after strength analysis exactly like other symmetric keys, as described above, and then used to do the encryption of the hidden plaintext, with the same nonce as the main message. When using a public key, the single ephemeral private key generated for the main message (Anonymous mode) or one's permanent private key (Signed and Read-once modes), is used to make an encryption key for the hidden message, by Diffie-Hellman combination with the special public key.

To extract the hidden message, the recipient must request that the padding included with the main encrypted message be itself subject to Decoy decryption, by checking the "Hidden msg." checkbox on the Options tab (named "Decoy" prior to version 2.4), or otherwise all that follows will be skipped. Decoy decryption is done before the main decryption to allow the hidden message to contain information that will help decrypt the main message. At this point the program displays a box asking the

user to supply the special key for Decoy decryption. PassLok starts assuming that this key is symmetric and simply uses it with nacl.secretbox.open and the same nonce as the main message in order to decrypt the hidden message. If decryption fails, PassLok tries again assuming it is an asymmetric private key, which makes the shared secret used for the encryption by combining it with the sender's permanent public key (ephemeral public key, in the case of Anonymous messages).

Decoy decryption fails in exactly the same way whether the special key supplied is incorrect or there is no hidden message to begin with. Thus, trial decryption cannot be used to detect the presence of a hidden message.

Secret splitting

PassLok includes a function to perform the Shamir Secret Splitting Scheme (SSSS). It uses a JavaScript library by A. Stetsyuk, with modifications by F. Ruiz to make it use the random number generator used by NaCl rather than its built-in RNG. When the user clicks the Split/Join button on the Main tab, PassLok first analyzes the contents of the Main Box so see if it consists of parts outputted by the SSSS function. If it does, it invokes the SSSS decoder, which requires a sufficient number of parts separated by linefeeds, and displays the result if successful or an error message if unsuccessful. Otherwise it calls the SSSS encoder. The SSSS encoder needs to know how many parts to make and how many to require in order to reconstruct the original; these are input via a special dialog that appears at this time. After the library finishes its calculations, the parts are displayed on the Main box, separated by extra linefeeds so they are easily separated from one another.

SSSS is based on polynomial algebra with big integer coefficients and is well explained elsewhere. The parts produced are not completely random, but it has been proven that it is mathematically impossible (not just infeasible) to reconstruct the original if one or more parts of the required number are missing.

Real-time chat

PassLok started supporting real-time chat with version 2.1. Unlike other communication programs, which rely on servers to pass the data between the participants, PassLok relies on direct connections between clients via the webRTC protocol, which is based on TLS at its core. Since TLS and webRTC are fairly complicated protocols that have been extensively tested for security, we will just assume that the connection is secure once it is established. This section, therefore, presents the method designed to establish the connection from PassLok.

PassLok is used to generate an encrypted "chat invitation" which, when decrypted, provides the secret information needed to connect to the chat session by webRTC. This information consists of the following:

1. Optional message: 43 characters where the sender can write the time for the chat and other short information.
2. Chat type identifier: "A" for text and files, "B" for text, files, and audio, "C" for text, files, audio, and video.

3. Chat room name: randomly generated sequence of common words (taken from the built-in blacklist), with common substitutions. The idea is to generate a name what would not reveal its being originated by PassLok. This string is padded with spaces to 43 characters.
4. Password: 43 base64 characters, also randomly generated.

The set is concatenated into a single string and encrypted with the currently selected encryption method. The sender is automatically added as a recipient, so he/she can decrypt the invitation like the other recipients. At the end, special tags are added to the encrypted item so the recipients can recognize it as a chat invitation.

When the item is decrypted, PassLok recognizes it by length (130 plaintext characters), triggering a special set of steps:

1. If there is a message, display it in a prompt and request confirmation to continue (it may be that the time set for the chat has not yet arrived).
2. Then load a special chat page in an iframe, and pass the rest of the decrypted material as a hash value (on the URL itself, following a "#" character). This chat page loads from an origin different from that of PassLok, so its functions do not have access to the functions and variables in PassLok. This is to prevent leaking secret information (such as the user's secret key) if the chat page, which has network access, becomes infected by malware.
3. PassLok rests at this point. Meanwhile the chat page contacts Firebase.io in order to obtain its external IP address, which is necessary to establish webRTC connections. It also checks whether a chat session with the given name (obtained through the URL hash) has already started. If it has, a "Join" button is displayed, otherwise it is a "Start" button, and additional instructions are shown to the user.
4. Participants are expected to supply a name to be identified during the chat session, and then click the button. The first participant gets a "waiting for other to join" message, while Firebase.io holds the IP address so it can give it to other participants joining that particular chat.
5. As more participants join, Firebase.io give them the IP numbers of the current participants, so they can connect directly via webRTC. Then webRTC negotiates participant-to-participant connections from each machine. The one starting the chat has control over who is allowed to join.
6. Before each new participant can actually connect to each of the others, the starting participant's machine must allow him/her. This will only happen if the correct password is sent during the negotiation stage. Notice that the password was never sent to Firebase.io, but rather remained secret with each of the participants.

Steps 2 and following can be repeated via a "Reset Chat" button, in case the webRTC connections to one of several participants fail. As of 2017, testing has shown that Firefox (as of v.35) provides the most reliable connection, while Chrome and Opera have problems connecting all of several participants to one another.

The webRTC connection does not have a provision for authenticating participants, other than the fact that they must have the correct chat room name and the correct password, which are obtained by successfully decrypting the chat invitation. Still it might be possible for one of the participants to extract that information from the invitation and give it to an uninvited third party, who then would be able to join the chat as a legitimate participant. To prevent this, it is possible to return to PassLok while a chat session is in course, and then to go back to it without needing to interrupt or reset the chat session. This allows existing participants to check credentials posted by new entrants to the chat. For instance, the existing participants can request newcomers to sign a random string, and then they can go to PassLok to verify whether the signature for that random string was made by the purported participant. To get around this process, the uninvited participant would need to possess another participant's private key.

Pad mode

This mode was added with version 2.3. The idea is to have a level of theoretical security comparable to that of a one-time pad, which has perfect theoretical security, meaning that it is impossible to distinguish the correctly decrypted plaintext from any other incorrectly decrypted "plaintext". The condition for this is that the key must have at least the same entropy as the plaintext, per Shannon's criterion. Normal computer encryption algorithms use keys of limited length (40 bits for DES, 128 to 256 bits for AES, and so forth), and so they don't have perfect theoretical security. Going through all the possible keys until one of them decrypts the message may take a long time, but it is a limited time anyway. With perfect theoretical security, one can go through all the possible keys (possibly of the same bit length as the ciphertext) and still have no idea of which of the plaintexts, each obtained with one of the keys tried, is the good one.

This is easy to do using a one-time pad, which contains one random character for each character in the plaintext: simply add each pad character to the corresponding one in the plaintext (there are many methods available to "add" two characters to obtain a third one). To decrypt, subtract each character of the pad from the corresponding ciphertext character. The part of the one-time pad involved in a given message must never again be used by either side. The problem then becomes one of generating the one-time pad (easy with computers), and of distributing it to both ends of the conversation, which is not easy at all.

PassLok gets around this problem by generating its "one-time pads" from common text, drawn from a book, a webpage, whatever, that is accessible to both the sender and the recipient and hard to guess by an enemy. Now, common text is not perfectly random, but it does contain some entropy, which has been estimated by several researchers (Shannon himself being the first) as about 1.58 bits per character, spaces excluded. PassLok uses the SHA512 hash function built into the NaCl library to extract this entropy by taking a chunk of the key source (encoded as a UInt8 byte array) long enough to contain at least 512 bits of entropy. Since UInt8 encoding uses one byte per character (8 bits), one way to do this would be to split the encoded plaintext into 512 bit chunks and then XOR each with the result of hashing a chunk of key source text of length 512 * 8 / entropyPerChar bits, where entropyPerChar is 1.58 (an adjustable parameter in PassLok).

The way PassLok does it, however, is slightly different. It first calculates the total length of key text needed using this formula: keyLengthNeeded = Math.ceil((textBin.length + 9) * 8 / entropyPerChar), where textBin is the plaintext encoded as a byte array (typically one byte per character), and takes a piece of key text of this length, which is good enough for the entire plaintext, not just a 64-byte chunk of it. It then determines how many 64-byte chunks of encoded plaintext there are. For each chunk, PassLok takes the entire encoded key text, appends a 9-byte random nonce and a byte-encoded index (0 to number of chunks-1), and then takes the SHA512 hash of the result. This is a 64-byte array than can be XORed to the corresponding chunk of the plaintext, resulting in the ciphertext, to which the nonce is appended in cleartext (plus a MAC, described below) before the whole thing is encoded in base64 for output. The process to decrypt is identical, with the ciphertext taking the place of the plaintext, resulting in the encoded plaintext.

A couple features allow the reuse of a given piece of key text multiple times. First, the random nonce causes the hash values obtained to be different each time (unless the nonce repeats, which is unlikely). And then, the starting point within the key text can be varied by the user, after PassLok prompts for it (default is the beginning of the complete key text, usually longer than the piece of it that is actually used in the computation), so that PassLok will go back to the beginning of the key text to keep drawing characters if it reaches the end before all the necessary key material (length keyLengthNeeded) has been collected.

We mentioned earlier that a message authentication code (MAC) can also be added to the ciphertext. This is to prevent an enemy from altering the ciphertext so it decrypts to something else, if he/she happens to know the plaintext at a particular location. A 16-byte MAC is made by drawing an additional piece of text from the key source, of length Math.ceil(16 * 8 / entropyPerChar). Start drawing at the point where the main encryption process stopped, wrapping to the beginning of the key text if necessary, then convert it to a Uint8 byte array, append the nonce and the encoded plaintext, and take the SHA512 hash of this. The first 16 bytes of the result is the MAC, which is appended to the ciphertext before output. To check the MAC, the recipient performs the same operation with the key text and the plaintext that was obtained from the decryption process, so the result should be the same as the MAC that is appended to the ciphertext.

This, of course, tells the decryptor that a particular plaintext is indeed the good one, which appears to destroy perfect theoretical security (it should not be possible at all to tell whether a certain “plaintext” is the correct one), but this is just an appearance. When computing the MAC, the decryptor is using new key material, which would have to be guessed as well if he/she/it is guessing the main key material. If nothing at all is known a priori about this new key material, then any particular MAC result can be produced with equal probability as guesses are made for the key material, so that the correct plaintext has the same probability of producing the MAC attached to the ciphertext as any incorrect “plaintext”. The question, then, is how much source entropy is necessary to achieve this. Clearly, 512 bits of entropy are enough, for in this case any particular output of the 512-bit hash function would have equal probability.

On the other extreme, 1 bit of entropy on the key source before the hash is applied (2 cases only) would produce only 2 different hash outputs (we are ignoring the entropy added by the nonce, which we presume has been fixed by the previous brute-force cryptanalysis), leading to only 2 16-byte MAC variations for a given plaintext (at most). An incorrect guess of the key material would cause a false negative on the correct plaintext half of the time. But if the plaintext is incorrect, using the correct (or incorrect) bit for the MAC will not necessarily produce the correct 16-byte MAC. In fact, the probability of this would remain at 2^(-128). In other words, the correct plaintext and an incorrect one would have very different probabilities of passing the MAC test (2^(-1) vs. 2^(-128)), so it would be possible to distinguish between the two plaintexts.

What then? 128 bits of entropy, same as the MAC itself? In this case there would be at most 2^128 different MAC values produced for the correct plaintext (the actual number would be smaller since the probability of all guesses leading to different 16-byte MAC values, without repeats, is very small), as we run all possible guesses for the entropy level of the key material used to make the MAC. So what is the probability of testing positive with the correct plaintext, as we make guesses for that key material? There is enough entropy to produce all possible MAC strings, but the probability of that happening is very small since that would involve no repeats, and therefore we should count on fewer than 2^128 different MAC values. If there are as many as, say, 2^127 different cases (just half of the maximum, which would mean that on the average every MAC value obtained appears twice), then the probability of testing positive with the correct plaintext is 2^(-127), which is double the 2^(-128) probability of testing positive with an incorrect plaintext. If the difference is not that great, it would still be detectable so long as it is not negligible, which is indeed the case.

In other words, so long as we can't assure a full entropy load for the key material used in making the MAC, we can expect the correct plaintext to test positive more often than an incorrect plaintext, which would cause a distinction between the two. Therefore, it is necessary to involve the maximum amount of entropy that a SHA512 output can have, which is 64 bytes.

As far as user interface goes, Pad mode is selected automatically if the key text entered in the key box is long enough given the length of the plaintext (at least 256 bits). On decryption, PassLok recognizes this mode through the single-byte identifier prepended to the nonce and the ciphertext before encoding to base64, which makes the base64 output begin with "d". When decrypting, even though any "plaintext" obtained is acceptable in principle, some byte combinations do not correspond to valid text. In this case the program displays a "Decryption has failed" message and skips MAC checking. Otherwise MAC checking is done, and if this fails the message rather says that "Message authentication has failed". This is still possible with an incorrect key source rather than a tampered message, but it may provide some information to the user.

Human mode

This mode was added with version 2.4. It implements a symmetric algorithm designed to be performed by a human with the help of pencil and paper, plus possibly a Tabula Recta printed beforehand. This is a Tabula Recta for English text, which comprises 26 different letters:

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
A	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
B	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A
C	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B
D	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C
E	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D
F	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E
G	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F
H	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G
I	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H
J	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I
K	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J
L	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K
M	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L
N	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M
O	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N
P	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
Q	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
R	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
S	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R
T	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
U	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T
V	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
W	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V
X	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W
Y	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X
Z	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y

As you can see, this Tabula Recta contains the alphabet 26 times, each row being the previous one shifted cyclically one character to the left. This makes the columns read the same as the rows. There is a header at the top and another on the left, but identical headers can be added to the bottom and right sides as well. The basic function of a Tabula Recta is to aid in letter addition. Just find the letters to be added at the top and left, and then follow the respective column and row. The letter at the intersection is the sum of those at the headers. For instance: K + T = D, S + F = X.

In human mode encryption, sender and recipient agree on a three-part key, comprising key 1, key 2, and a seed mask. Key 1 is used to generate a mixed alphabet (letters A to Z, but in mixed order) that will be placed on the left header. Key 2 is similarly used to make a mixed alphabet for the top header. The seed mask is used to encrypt a random seed made of as many letters as the mask, different for each message. The process, which is explained in more detail on this post at prgomez.com, goes as follows:

1. Take the plaintext and remove any accents and diacritical marks so the alphabet in use is the standard 26-letter alphabet. Then replace any Q with K, and any punctuation other than commas, and all spaces (reduced previously to single spaces), with Q. This is so that spaces and the most essential punctuation are preserved. Encode numbers as A,B,C,etc. Write the result as a row.
2. Generate a mixed alphabet for the left side of the Tabula Recta from key 1, and another for the top from key 2, as follows:

a. If a key is longer than 25 letters, if needs to be compressed first: write it, one letter per cell, in a table containing 25 columns. When the first row is filled, continue filling the second, and so on until all letters have been written. Then do the following for each resulting column: look the first letter on the top of a straight Tabula Recta (alphabets on the edges are not mixed), then down that column until you find the second letter (if there is one), then left or right until you find the third letter (if there is one), and so on until the last letter is found, and then again perpendicularly to read of the result at top, bottom, or one of the sides. Write down the result for each column, and then you get the compressed key.
b. Take each compressed key and on the corresponding header of the Tabula Recta write down new letters in the order they appear on the compressed key; if a letter in the text key has already been written, write instead the first letter before it in the alphabet that is still available (wrap around to the end if needed); then write the rest of the alphabet in reverse order. From now on, we will look up letters on these rather than of the original headers.

3. Take the seed mask (processed like the plaintext, if necessary) and write it immediately to the left of the plaintext. Then write a randomly chosen letter below each letter of the seed mask. This is the seed, from which the keystream will be generated.
4. Extend the seed into a keystream so all spaces below the plaintext are filled, this way: Look up the first keystream letter still available at the top of the Tabula Recta, then down until you find the letter that follows it in the keystream, then go left to read a letter on left alphabet, which you will write in the next available position on the keystream. Mark the first keystream letter you looked up so next time you start with the next letter.
5. Now do the following for each pair of letters consisting of a plaintext letter (plus seed mask) and the keystream letter right below it: Look up the plaintext letter at the top of the Tabula Recta, then go down until you find the keystream letter, then left to read a letter on the left alphabet, which you will write below the pair of letters you involved in this operation, forming the ciphertext.

The decryption process, starting from the ciphertext plus the same three-part key is identical for steps 1 and 2, involving keys 1 and 2, and the Tabula Recta, and then it goes like this:

3. To extract the original seed, write the seed mask directly below the beginning of the ciphertext, one letter per space, then do the following with each pair of ciphertext and seed mask letters, read vertically: look up the ciphertext letter (top) on the left of the Tabula Recta, then go right until you find the seed mask letter, then up to read the seed letter at the top header. Then write it below the pair.
4. Now extend the seed into a keystream so all spaces below the ciphertext are filled, using the same process as when encrypting: Look up the first keystream letter still available at the top of the Tabula Recta, then down until you find the letter that follows it in the keystream, then go left to read a letter on left alphabet, which you will write in the next available position on the keystream. Mark the first keystream letter you looked up so next time you start with the next letter.

5. To recover the plaintext, we use the same process as step 5 when encrypting, except that we do it as in step 3 above, that is: look up the first letter (ciphertext) on the left side of the Tabula Recta, then right until we see the keystream letter, and finally up to read the plaintext letter at the top header.
6. The last step is to recover spaces and punctuation. Replace every Q with a space, or a period and a space if there are two Q's in a row; replace every instance of "KU" followed by a vowel with "QU". The result won't be exactly the original plaintext, but it will be quite readable.

It seems a little involved at first, but it really isn't so when compared with any computer-based encryption algorithm. And yet, the algorithm is remarkably strong for the effort, with ciphertext that is indistinguishable from random letters for messages a few tens of thousands of characters long. This is a lot longer than anyone would want to do by hand.

The potential key space is quite large. Each of the mixed alphabets has 26! possibilities, which is equivalent to 88.7 bits each (keys 1 and 2). The seed is actually random and of unlimited length. If seed length is, say, also 26 characters, then there are 26^26 possibilities, which is equivalent to 122.6 bits, for a total of 300 bits. Tests have shown that a single mistake in either key or the seed mask (and therefore the actual seed) produces a "plaintext" that appears to be completely random, so it is not possible to reach the correct key by gradual approximation, even if the seed length is known.

The lagged-Fibonacci method used to produce the keystream is not yet well understood, so occasionally a given seed might produce a keystream that does not pass randomness tests, but this can be detected so that a new seed can be chosen. When the algorithm is done by hand, the sender will see it happening and will be able to change the seed. When done by computer, a test is performed so a new seed may be chosen if the test fails.

PassLok implements this algorithm in JavaScript. It is triggered on encryption when the string contained in the working key box (where the recipient's Lock or shared Key are placed) is made of three alphanumerical strings separated by single tilde characters ~, that is: key 1 ~ key 2 ~ seed mask. Spaces at the start and end of each string are ignored. On decryption, it is triggered when the ciphertext contains only letters (no special initial character), and an error message is displayed if the key string does not conform to the above format. The randomness tests performed on the keystream (encrypt only) to determine is apparent randomness are two: chi-squared test on single letters, and chi-squared independence test on pairs of consecutive letters.

Error correction (removed as of version 2.2)

In order to make PassLok as accessible as possible, some of its output (ezLocks, for instance) can be transmitted via voice rather than as written text. But this has the danger of introducing transcription errors. Similarly, if a PassLok item is printed and then OCR'd back to digital format, it is likely that errors will occur. There can also be errors due to the email or texting program introducing linefeeds or extraneous characters where they don't belong. This is why PassLok has two types of error correction.

One of them is the Reed-Solomon scheme, the same algorithm used in optical media, bar codes, and QR codes. The other is a simple best-of-three algorithm.

The Reed-Solomon algorithm (RS) involves making a digest or hash of the data, which is appended to the data as part of the tags. Unlike common hashes and checksums, this digest allows the program to detect not only if there has been a change, but also its location and what the correct value should have been, up to a maximum number of changes. The details of the RS algorithm are rather complex and the reader is referred to sources that can explain it better than this document.

Users can easily spot where the correction code begins in a PassLok output item because it is separated from the main data by an "=" sign, same as the tags at both ends. The error correction code is also hexadecimal rather than base64, which helps in spotting it. PassLok uses the erc-js Reed-Solomon error correction JavaScript library by cryodex, as found on GitHub, in order to generate the code and later check the data against the code. These are the parameters used in PassLok:

Number of erasures to be corrected per 256 characters	10
Characters implicitly considered to be correct	128 (ASCII set)
Number of errors to be corrected per 256 characters	5
Code length per 256 characters	20

RS code generation is called after the item is generated by PassLok (excluding the tags, which are added later). It works on blocks of 256 characters, so the length of the RS code depends on the length of the item, with 20 RS code hex characters being generated for every 256 (potentially ASCII, but actually part of a 73-character set) characters of the output data, or less. All kinds of output items have RS codes added by default: Locks (public keys), encrypted material, digital signatures, parts produces by the SSSS. If this is not desired, users can prevent code addition it by checking the "no Tags" option. Items without RS codes can be processed, although without the benefit of error correction.

RS code verification and possible correction takes place right after a function is called on a PassLok item: the RS code is separated from the main data and the verification subroutine is called. Characters that are not in the 73-character set used by PassLok are replaced by a "ã" character that is not in the ASCII set and the subroutine recognizes as invalid, prior to calling the RS verification function. This is because the limit for "erasures," that is, characters that are not valid, is double than for valid (but wrong) characters. In the case of Locks entered into the directory Edit dialog, error correction takes place as soon as the user pastes (or finishes typing) a Lock, to make sure only correct items are entered in the directory.

RS error correction relies on the data string being of the correct length, without additions or deletions (not the same as erasures). If some data is missing or spurious characters have been added (or there are more errors than the limit noted above) the RS algorithm will fail. In this case, PassLok displays a

message instructing the user to strip the correction code and try again. When the RS code is missing or does not have the right length, PassLok simply strips it without calling the RS correction algorithm.

Since the error correction algorithm is slow compared with encryption and decryption, PassLok displays a message asking for user confirmation to generate or check the RS code if the ciphertext is longer than a certain amount. This length is currently set at 100,000 characters. If the user cancels at this point, code generation or checking is skipped without affecting the rest of the algorithms.

For very noisy channels where data might disappear or noise be added in chunks, PassLok includes an optional error correction method, based on a best-of-three selection, that can handle this situation. To use it, the sender only needs to copy the item three times, separated by "@@" markers. This happens automatically if "Triple" is selected in Options. On the receiving side, PassLok scans for items that are tripled like this (regardless of whether "Triple" is selected or not), splits them into three parts, and does the following before doing anything else, for each character in sequence:

1. If all three copies have the same character, accept it as good and move on.
2. If two copies agree and one disagrees, accept the one with agreement.
3. If step 2 is taken, perform a check of the following character on the string that was different. If it is the same as the accepted character, that means the different character was an addition, so delete it. If the following character on at least one of the other two strings is the same as the different character, that means a character was deleted, so add the accepted character to the different string at the current position.
4. Steps 2 and 3 can be applied recursively so that additions or deletions consisting of several characters in a row can be corrected.
5. If at some point no agreement can be found between two strings, stop and report a failure. The user can always select which of the three parts to keep, or try to correct errors manually.

After best-of-three is applied, the normal process resumes, beginning with RS correction of the accepted string if an RS code is present.

The heading of this section states that this feature was removed starting with version 2.2. The reason is that the error-correction process is quite slow compared to the encryption or decryption process. Since most encrypted messages are likely to be transmitted digitally and those channels already use error-correction strategies, adding error correction to PassLok seemed redundant, especially since it takes so much computational effort.

Threat Analysis

The scenario presented earlier is just one of the ways in which a third party could try to subvert the security provided by PassLok. Here are a few more attacks, presented here as a simple eavesdropper (Eve) or powerful enemy (Mallory) trying to mess with communications between Alice and Bob:

1. Man-in-the-middle: Mallory posts fake public keys for Alice and Bob and manages to have them use those keys as he intercepts all communications between them. Thus Alice encrypts information with "Bob's" public key (actually Mallory's), which Mallory intercepts, decrypts, re-encrypts with the authentic public key for Bob, and sends to Bob. He can do the same thing for information traveling from Bob to Alice.
2. Rubberhose: Mallory kidnaps Alice and forces her to reveal her private key without Bob knowing anything. Then he can read their correspondence and implicate both Alice and Bob.
3. Dictionary: Eve grabs Alice's public key and runs complete dictionaries of trial private keys through an optimized program (possibly with optimized hardware as well) that generates public keys for each trial key until a match is found. Then she knows Alice's private key and can impersonate her any time.
4. Rainbow table: As above, but Eve performs the calculations ahead of time and stores the public keys matching every private key tried. Then she only needs to look up any public keys she wants to crack and find the private key from which it proceeds.
5. Cross-scripting: Mallory directs Bob's browser to fake servers, which he controls. The servers send modified web pages containing code snippets that change the PassLok code, so its security is undermined, or steal sensitive data such as Bob's private key.
6. Tracking: Mallory monitors external Internet resources so that, as soon as one of them is accessed, he is able to recognize the request as coming from a PassLok user. He then grabs the IP number, which gives away the user's location.
7. Timing/side channel: Eve is subjecting the correspondents' computers to indirect surveillance by recording the time it takes for messages to travel between them, or by recording sounds, lights, heat, electrical impulses other than those encoding the messages, which give away information about the processing.
8. FISA court order/server hacking: Mallory presents a court order to the web host of the PassLok code, asking access, plus a gag order against revealing anything, or simply obtains unauthorized access to the web server by hacking into it. Then he changes the code in subtle ways to undermine its security and waits for Alice and Bob to download it and use it. From then on, he can decrypt all their PassLok-encrypted communications.
9. Zero-day: similar to the attack above, except that the weakness is in the browser, the operating system, or the hardware, planted beforehand so it can be used when needed.
10. Hardware theft: Mallory steals Alice's computer, which contains all her PassLok data, hoping to impersonate her and communicate with Bob.
11. Data theft: Bob has been using a computer to which Eve also has access. When Bob steps out of his cubicle to get a cup of coffee, Eve quickly steps in and copies all PassLok data stored in Bob's machine.

Surely there are many more attacks, but these are the most common, and the ones to which PassLok is most vulnerable. Let's see how PassLok tries to foil them.

Man-in-the-middle

The issue here is user authentication. Alice cannot tell that Bob's public key is actually his just by looking at it, even if contains a label saying so. This problem is shared by all public key cryptosystems, and dealt with in various ways. A popular one is to have a third party sign the public key with his/her/its private key, which would be proof that this third party believes the public key is authentic. This third party can be a person that both Alice and Bob know and trust, or a stranger that is trusted implicitly because of its title or position. If a known person, we are talking about a "web of trust," if a stranger, we have a "certificate." Webs of trust have not progressed much since they were proposed more than twenty years ago in order to provide some authentication to PGP public keys. The natural offshoot of this is Certification Authorities (CA), which are known to the browsers from factory and trusted implicitly. CAs are at the core today's security protocols like SSL/TLS. Unfortunately, there have been reports of CAs not following due diligence before issuing certificates or just plainly lying (a number of CAs are government-operated). If this is an indication of the current state of affairs, a recent NSA report does not list SSL/TLS as a major obstacle to its information-gathering efforts, while PGP and some protocols proper to Apple systems are listed.

PassLok does not even attempt to create a web of trust or sign certificates. Instead, it relies on rich media for user authentication. PassLok Locks (public keys) are short enough that users can actually dictate them over the phone or in a short video. In either case, those viewing the media can associate the Lock being read with the voice and the face of a person they recognize, and be sure of the Lock's authenticity without a need for witnesses. Locks are equally functional by themselves or with a URL (possibly pointing to an online video) appended to them, and so it is recommended that users do make these videos and append their URLs to their Locks before they publicize them. One of the Lock versions, ezLoks, is designed so Locks are very easy to read without ambiguity by re-coding them to base36, which does not distinguish between lowercase and capital letters. Specific instructions for making authentication videos are contained in the PassLok help system.

In addition to this, the help system also contains instructions for using the well-known interlock protocol, which basically involves splitting a special encrypted message into two parts that are sent at different times. If the protocol is followed correctly, a man-in-the-middle would be forced to act on the contents of the split message before the two halves are available, which is unlikely to not be discovered.

Rubberhose

This has already been discussed when Decoy mode was presented. The concept here is "plausible deniability." One way to achieve this is to encrypt two different messages, the real one and a dummy, innocuous one, encrypted under different keys. If a powerful attacker demands the decryption key, the victim can give him/her/it the key that decrypts the dummy message while keeping secret the key that decrypts the hidden message. This is likely to succeed if the presence of the hidden message cannot be detected. PassLok implements a form of this in which one message can be detected and the other

cannot, by encrypting the hidden message into a "padding" string that would otherwise be completely random (or rather, pseudorandom). This process is optional, selected by a checkbox on the interface. In versions prior to 2.2, the padding string had a use as salt value in computing encryption keys, but this is not really necessary and so starting with version 2.2 the padding string is there only as a possible container for a hidden message—which doesn't mean there is one. Most likely, the padding string is the result of encrypting a random string with a random key.

This is actually better than always containing two messages. If this were the situation, the attacker would know that the victim must be forced to produce two keys rather than one. But by making the feature optional the attacker can never be sure that there is a second message encrypted under a different key, and the victim could plausibly deny its existence.

Another feature that is related to this attack is deniability, defined as the impossibility to link an encrypted message to a particular person, whether as sender or as receiver. In some circumstances, just the fact that an encrypted message has been exchanged between two people can be considered incriminating, regardless of its content. The problem may surface as soon as the message is sent (external deniability) or, in some cases, after one of the parties has been forced to relinquish his/her private key (internal deniability).

All the encryption modes in PassLok possess external deniability, as care has been taken to ensure that no part of them leaks the correspondents' identity to those not possessing the necessary keys. This is better than in other standardized programs, such as PGP, where the standard calls for the recipients' public key ID's to be included within encrypted messages. The trickiest part of providing deniability is encountered when making the ID tags that PassLok uses to signal the location of the message key, as encrypted for a particular recipient. To avoid leaking the recipient's identity, the ID tag is made by encrypting a piece of identifying data, such as the recipient's public Lock, with a shared secret that only the recipient can synthesize starting from his/her private key and an ephemeral or permanent public key.

After the recipient's private key is compromised, the Anonymous mode ID tag leaks the recipient's identity (trivially, since the message itself can be decrypted), but not the sender's. Signed mode ID tags also leak the sender's identity in the sense that trying his/her Lock would result in a successful decryption once the recipient's private key is compromised. Read-once and Read-once ID tags, except for the first message in a new series, remain impervious since the ID tags are encrypted with ephemeral keys as well as the message itself.

Dictionary

This also has been discussed already, when describing how public keys are derived from private keys. One common way to crack a key, starting from a known public key, is by trying likely choices obtained by combining words in special "hacking" dictionaries, until one of them generates the public key. This is much faster than trying every possible combination of valid characters, and typically succeeds in cracking 90% of real passwords within a couple minutes of computer time, if only a one-way hash of it is known. Public keys are at least one order of magnitude slower to generate than hashes and the specific

speed depends on the type of public key, but the process is still within the reach of even amateur hackers.

PassLok combats this by adding extra iterations of the SCRYPT key-derivation function for lower values of key entropy in order to multiply the time required to run through the worst key choices, as was discussed earlier. In this section, we describe how the entropy is measured. The process for English language keys is as follows:

1. Remove all spaces from the string being evaluated.

2. Detect the types of characters used, and add the total number of possible values to a counter. For instance, if numbers are present, add 10 to the counter; if small case letters, add 26; if capitals, add another 26, and so on. The result will be called Ncount.

3. PassLok includes two dictionaries: one contains the 1,000 most common English passwords, the other contains the 10,000 most common English words. In order to account for common substitutions (“1” instead of “i”, “3” instead of “e”, and so on), perform those substitutions on the string (the dictionaries already have those substitutions made).

4. Then remove every substring that is found in the common password blacklist. Those will get no credit.

5. Then find every substring that is found in the regular word list. Count how many distinct ones are present, Nwords, and remove them all from the string. What remains should be text containing no words from the dictionaries.

6. Now remove characters that are repeated, or are repeated periodically. This can be done in JavaScript with this command: string = string.replace(/(.+?)\1+/g,'$1')

7. Finally, count the number of characters remaining, N, and perform this calculation:

   entropy = (N*log(Ncount) + Nwords*log(wordlist.length + blacklist.length))/log(2)

   where wordlist.length and blacklist.length are the numbers of entries in the common word and password dictionaries, respectively. The result of the calculation is the entropy of the original string, in bits. The essence of the calculation is to find the fraction of the total spaces of words and single characters that are used in the given string, and sum the entropies due to each. Blacklisted words add to the size of the dictionary but give no entropy credit.

Users get to see what the result of choosing a Key with a low entropy will be before they commit to it, because PassLok displays the time necessary for processing (mostly SCRYPT iterations) if that Key is chosen. In order to give an accurate number, PassLok times the calculation of 1024 iterations of SCRYPT with a dummy key and a dummy salt, for the usual parameters, done 10 times over. This is done as soon as PassLok loads and the result, which will vary according to the computational power of each particular device, is stored as a global variable. Calculating processing time for a given Key is done with this

formula: time in seconds = storedTimingInMilliseconds/10240000*iterations*2, since key derivation is done twice for a given user-supplied Key: once for the key that decrypts locally stored items, and once again for the actual private key

Rainbow table

A hacker with access to large computing power and storage might pre-calculate the public keys resulting from all the entries of a hacking dictionary (and its common variants) as user Keys. The result, called a rainbow table, would mean that cracking a Key would be as simple as looking it up on the table, to see if its published Lock is there. This is a real threat for public-key systems like PassLok, where user-supplied private keys are used rather than pseudorandom keys.

To combat this, PassLok encourages users to enter their email or some other piece of non-secret, but personal information. This string is used as salt by the SCRYPT key-derivation function, so the resulting key depends from it. Now, it is highly unlikely that those making a rainbow table have decided to add precisely this user's email (or whatever) when they made it, for in this case the table would only be able to crack Keys belonging to this person. This means that a user's Key will only be found in a pre-cracked rainbow table if no personalizing salt value was used.

PassLok accepts anything as personalizing salt, and will store it so the user does not have to enter it again. Before it is stored, it is encrypted with the result of stretching user's Key, plus his/her user name as salt, and a fresh random 9-byte nonce. The private key itself, whether stretched or not, is never stored. When the user enters his Key into PassLok, it is first stretched using the user name as salt, than the result is used to decrypt the stored salt value, and the Key is again stretched with this salt value to make the Ed25519 private signing key, from which the Curve25519 DH private key is immediately computed. The process ends with the user's Lock (signing public key, encoded in base64) being calculated. All the private values are stored in global variables that are overwritten after five minutes of private key inactivity.

If a user desires ultimate security, the personalizing salt can be a random value. PassLok has a button that will display a 43-character base64 string (256 bits) when the personalizing string is requested. Since it is stored in encrypted form, the user does not have to remember it. Since the stretched key used in generating the actual public key depends on a long random value, it is impossible to crack except by brute force, which is also essentially impossible given the bitlength of the random value, but this results in a loss of portability since entering a new random value (presumably at a different machine) would result in a different public key.

Still, this random token can be backed up into an encrypted string, along with the other permanent settings, so the user may take it to another machine. The Chrome app version of PassLok does this automatically through the Google servers (again, everything is encrypted before it goes anywhere), so users only need to log into their Google accounts in order to find all their settings, including the random token, synced to a machine they've never used before.

Cross-scripting

Every html document that loads data produced by a third party is subject to cross-scripting attack, where the data loaded contains malicious JavaScript code that is executed automatically. The effect can be as subtle as a small change in the cryptography primitives that renders them insecure, or as blatant as reading the user's secret Key from memory and sending it out somewhere. In either case, the user may not be aware of anything being wrong. This is the main reason why a number of security experts insist that cryptography cannot be made secure so long as it is based on JavaScript.

PassLok combats this attack in several ways:

1. PassLok contains no instructions that would imply a connection to a server, with a few well-defined exceptions:
    a. Video tutorials and additional documents: all of these are links on the Help tab, which open a separate window on the browser (a separate app, in the iOS version). All links are TLS-based.
    b. General Directory: this is a page running at www.passlok.com, which executes some PHP code remotely. The page runs within an iframe that always has a different origin from the main PassLok code, so browsers forbid communications between the two codes other than by POST commands. Those commands are used to pass public keys back and forth, which are analyzed for validity and displayed for the user to see inside text area elements. Since public keys are relatively short strings, it is unlikely that malicious JavaScript code might be put into them without the user seeing it.
    c. Real-time chat: the code for this also runs within an iframe served from www.passlok.com, so it's a different origin from the main PassLok code. In this case, no data is passed by POST commands. The only communication between the two codes is when PassLok passes the chatroom code and password to the chat code, which is done through the URL of the chat page.
    d. The Chrome app version of PassLok syncs the contents stored in the local directory through Google servers. This is done using specific instructions that are validated at both ends, using TLS connections at all times. All private items in the local directory are individually encrypted with the user's secret Key, plus the user name acting as salt, before they are synced.
2. All strings that might be user-generated, such as just-decrypted plaintext, are passed through a strong white-list filter that deletes all HTML tags not due to the editing capabilities of PassLok itself as soon as the string is decrypted (JavaScript instructions are bracketed between <script> and </script> tags). The idea is that, even if someone manages to slip malicious code into encrypted items, the code will be stripped off before it has a chance to execute.
3. PassLok contains no instructions, such as EVAL, that would interpret other objects as code, or any in-line code contained in HTML elements. This is a good practice in general, and is enforced by Google for apps served through the Chrome store.

Tracking

PassLok limits the ability of a powerful enemy to track users by not connecting to servers for its proper functions. If a user connects to a server for sending or receiving an email, for instance, he/she does so by means of a separate application, not through PassLok. Mallory, the powerful enemy, has no way to know that the email was encrypted by PassLok unless the message itself is analyzed, and PassLok has steganography functions to make this as difficult as possible.
But PassLok does contain a few links, and loads some external pages into iframes, which could potentially lead to leaking the user's IP number:

1. Instructional videos linked from the Help tab are hosted on YouTube, which receives millions of requests every day. The videos do not load into PassLok, but into a separate tab or window on the browser, and are accessed via SSL/TLS only. Mallory would have to gain access to the YouTube server, and obtain the IP addresses of requests to view those particular videos. While this is not evidence of PassLok use, it would reduce the number of false positives considerably. To somewhat reduce this risk, the Help tab contains a statement warning the user that watching a video could leak his/her location.
2. The General Directory is hosted at https://www.passlok.com/lockdir. This page is loaded into an iframe when a button is clicked. Similarly as with the YouTube videos, Mallory would need to gain access to this server, which is under the control of the PassLok developers. Alternatively, they could seek access through the web host, without the developers' knowledge. Use of the General Directory is not evidence of PassLok use, but the number of false positives would be small. Because of this, PassLok warns the user of a possible IP leak via a tooltip on this button, plus a statement in the corresponding Help item.
3. The chat function loads a page from https://www.passlok.com/chat into an iframe as soon as an invitation is decrypted. Therefore, what has been said in the paragraph above can be said of this function. Users are warned the moment they open a chat invitation and are offered to cancel before the page is loaded.
4. Additionally, establishing a chat connection involves contacting a server at Firebase.io. Anyone who can view the names of the chat rooms set up on this server might be able to identify one originating from PassLok, and might thus obtain the user's IP number by attempting to join the chat, even if the connection does not succeed. This is especially dangerous since the chat connection will fail to be established if an anonymizing tool is being used. PassLok attempts to combat this situation by asking the chat originator for a nondescript name for the chat, which would be hard to pick out among the thousands of chat rooms existing at any time on this server. If the user does not provide a name, PassLok synthesizes one by picking one or two words from its built-in blacklist of common passwords.

Timing/Side channel

It is suspected that the time it takes for many elliptic-curve operations to complete is related to the numbers involved in the operation. Therefore if Eve the watcher knows this time, which possibly she can deduce from the timing of the messages exchanged between Alice and Bob, then she knows something. After collecting enough information, she may be able to guess one of the secret keys within less time

than it would take to do a full brute-force or dictionary search. A similar thing can be said about other non-message information collected from Alice's or Bob's machines.

This is a real problem for cryptography used between computers, as in SSL/TLS, because they are on fixed, predictable timers, machines talking with machines, but PassLok does not operate this way. PassLok does not transmit anything automatically; instead, its output is presented on a graphical interface so the user can copy and paste it into the program that will actually do the transmission. It would take a user with superhuman powers to do this in a way that the natural variability due to manual operation would not completely obfuscate the timing information. In addition, the elliptic curve operations in NaCl have been designed specifically with the criterion that they should always take the same amount of time to complete.

Court order/server hacking

The PassLok code is served from shared public servers administered by third parties. It is conceivable that a powerful attacker, such as a government agency, might gain access to those servers by legal means and alter the code at its source so that security is compromised without the users' knowledge. Of course, any code delivered by an Internet source is vulnerable to this attack, as well as to the similar attack where hackers gain unauthorized access to the source server and are thus able to modify the code.

This is perhaps the attack to which PassLok is most vulnerable. In order to combat it, we are following this strategy:

1. PassLok is served from multiple servers, at different locations in the world. The majority of these locations are within US boundaries, but at least one is outside the US and controlled by a group that is unlikely to comply to a US-issued legal command. Another source (GitHub) displays the source code at all times and track changes and alterations by means of checksums. All sources serve the code through TLS. PassLok enforces TLS connectivity by re-writing the URL if connection is attempted by regular http. Sources are monitored for changes on a constant basis via third-party scanning bots.
2. Secure hashes are taken of every new version of the genuine PassLok code, and those are posted on *different* servers from those delivering the code. Users are encouraged to check the hash before using PassLok. An item on the Help tab gives simple instructions for doing this. To defend against the possibility of someone being able to change the code at all servers as well the hashes at all locations where they are posted, a video of the PassLok author (yours truly) reading and displaying the SHA256 of the current version is also prominently posted. This video is protected from tampering by playing a well-known piece of music in the background.

Zero-day

But what if the attacker has compromised the browser, or the operating system, or even the machine itself, even before PassLok is loaded? Don't many public computers have keyloggers installed by their owners for liability reasons, just to mention one likely pitfall?

In this case, PassLok's defense is its portability. It can run from file as well as from a server, and it does not need a particular browser to function properly. A user that is concerned with zero-day vulnerabilities would not use the installed OS, but rather would boot the machine from a USB drive under his/her control, which contains a portable OS that is open-source or otherwise trusted. Operating systems that have worked well in our tests are Puppy Linux (open source) and Liberté Linux (not open-source, but security-oriented). Both OS's can save sessions back to the USB drive in encrypted form, in case an attacker gains access to them. Then the user would run a local HTML copy of PassLok that has been previously checked for authenticity, rather than rely on one downloaded from a server. If hardware keyloggers are a concern, both versions of Linux include on-screen keyboards that can be used for the most confidential parts of the workflow. Hardware screen loggers are unlikely to be installed undetectably.

Hardware or Data theft

But what if my trusted computer or mobile device is stolen, or simply an attacker gains temporary access to it so that he/she/it can copy whatever PassLok has saved locally? This is a particularly pressing concern for machines that are routinely shared by several people. If the first incident happened to a computer where PGP had been installed, just to mention one popular program, I would lose the use of my private key if I had not backed it up, so that I'd be forced to issue a key revocation certificate and come up with a new key. If would be a little better with simple data theft, since private keys are always stored encrypted, but then the attacker might be able to obtain my private key by brute force or dictionary attack on the encrypted key, which would not take long to succeed if my encrypting passphrase was weak, and I'd never know.

Here's what PassLok does to foil these attacks:

1. In PassLok, the user's secret Key is *never stored* anywhere permanently. This secret Key resides only as the value of the text box where it is input at the start of the session, and as global variables after the Key has been stretched. PassLok keeps a special timer that tracks the last time when a version of the secret Key was used, and deletes all the versions of it existing in memory after five minutes of inactivity (unless the user manually disabled this feature when the key was input).
2. PassLok does offer the possibility of using a random token to strengthen the user's key, and this token is saved permanently to the local directory, but it is first encrypted by the user's key before it is stored. The Chrome version exposes the encrypted token to Google, so that attackers who compromise Chrome sync would have access to it without having to gain access to the user's machine. In this regard, PassLok has the same vulnerabilities as PGP and other programs, except that using a random token is optional rather than obligatory.
3. PassLok stores symmetric and ephemeral keys associated with other users, also encrypted by the user's secret Key (stretched with his/her user name). An attacker who obtains those could guess the secret Key by brute force or dictionary attack, and then he/she/it would have the secret Key as well as the contents, including the random token if there was one. Since those items are simply encrypted with XSalsa20, the process would be much faster than trying to

reverse a public key, but again, obtaining the items themselves would be much harder than obtaining a user's public key. We consider the difficulty of either attack to be comparable, and rely on the strength of XSalsa20 augmented by the variable key stretching process mentioned earlier (which increases computational expense for weaker keys) to protect against it. In order not to leak even the length of an item (or the fact that the stored salt is a random value), stored items are padded with spaces so they are at least 43 characters long, before encryption takes place.

4. In the event of a user deciding that a machine is no longer to be trusted, the entire local directory can be erased with a single button press. The process produces a backup item on screen, which can be used to restore the local directory on a different machine. This backup is encrypted by the user's key, in addition to private items contained inside being individually encrypted in the same way. If the Chrome app version is used, PassLok additionally offers to delete the items synced to Google so that nothing is left behind.

Steganography methods

Text steganography

Steganography is the art of hiding, rather than encrypting, private information. The difference with cryptography is that the information is undetectable rather than unreadable. Steganography is desirable for certain scenarios. For instance, people wish to correspond in a country where cryptography is illegal. If cryptography alone is used, its random-looking output alerts the authorities of the fact, with possibly bad consequences.

Much of today's surveillance is being done by automated scanning programs or "bots." They are very fast and never tire, but since they are not human it is hard for them to catch whether what people are saying to each other actually makes sense. This is the basis of text steganography, where random-looking strings can be converted into apparently normal text (at least, as far as a bot can tell), and back on the other end. PassLok implements five kinds of text steganography. In addition, PassLok implements one kind of image steganography, where the secret text is hidden in the least-significant values of pixel data in a cover image, resulting in an image that human eyes cannot distinguish from the original.

These are the four kinds of text steganography implemented in PassLok, none of which requires the recipient to possess the cover text that the sender used for encoding:

- Words: Characters are encoded as words taken from the cover text, two for each character.
- Spaces: Characters are encoded as spaces between words of the cover text, and other than this the cover text looks the same.
- Letters (default): Some characters in the cover text are switched between their normal "Latin" forms, and other forms (Greek, Cyrillic) resulting in text that looks identical but is able to encode a text.
- Sentences: Each character of the original is replaced by a sentence from the cover text. The sentences are for 11 different lengths and end with one of six different punctuation characters.
- Invisible: The entire string is encoded as characters that won't take any space on the page. The invisible string result is bracketed by two dummy strings that can be modified at will.

As a table:

Method	**Spaces**	**Punctuation**	**Grammar**
Words	constant	random	random
Spaces	variable	original	original
Letters	variable	original	original

Sentences	constant	encoded	correct
Invisible	variable	none	none

None of these methods is perfect, since the Words and Sentences methods produce output that, even if grammatically correct, does not really make sense, and the spaces in Letters and Spaces output are somewhat irregular so that a keen eye can detect there is encoding. But it is likely that one of them might be able to defeat a particular scanning bot that is not looking specifically for text encoded by each method. To a human observer that is not paying much attention to what the text says, the last three appear quite acceptable since the sentences are grammatically correct.

When each method is invoked via a button on the interface, PassLok first analyzes the string to be encoded and determines whether it is base64 and, therefore, likely to be an encrypted message. Spaces, periods, and commas, for instance, are not in this set, so that any text that contains any of these will not pass the test. If the test fails, PassLok understands the text to have been already encoded and then tries to identify the method used, no matter which button was pressed to initiate the process, and then decodes it. If the test passes, on the other hand, PassLok encodes the item using the method corresponding to a selection made in Options. We now proceed to explain the details of each method.

Words

In this method, a list of distinct words is made from the cover text, and then the words are classified in groups according to their length mod 8. Here are the steps for encoding:

1. Clean the cover text of any non-literal characters, reduce all letters to small case, and break it up into separate words in an array. Remove any nulls and repeated items. Then make an array of arrays, where each sub-array contains all the unique words having a particular length (mod 8, so that 10-letter words would be grouped with 2-letter words, and so forth).

2. For each character in the string to be encoded:

    a. Take the index of the character in the key alphabet and express it in base 8. Since there are only 64 different characters, this will result in two base8.

    b. Take a random word from the array made in step 1, having the first digit length (mod 8), and a second word having the second digit length.

    c. Add spaces as necessary plus a random period, comma, or nothing according to set probabilities.

3. Capitalize the initial letter and after periods. Add the final period.

To decode the result:

1. Turn into lower case, and strip periods, commas, and linefeeds. Break it up into separate words, making an array.

2. Then for each pair of consecutive words: measure their lengths (mod 8) and convert the resulting base 8 number to decimal. Then write the character at that position in the key alphabet.

Spaces

Since all characters in an output string are base64, they can be represented by 6-bit binary numbers. These numbers, in turn, are encoded as single or double spaces between words of the cover text. Here is the encoding process:

1. Take the cover text and break it up into words as described above, but this time preserving punctuation and original capitalization. Then record the first word.

2. For each character in the original string, represent the base64 character as binary, and then:

    a. Write a single space for a 0, or a space followed by an invisible space (code 200c in JavaScript) for a 1

    b. Then write the next word (plus punctuation, if any) of the cover text.

3. Add a final period.

And the decoding process:

1. Take the text, strip linefeeds, and count the spaces between words, making a binary code where "1" represents a double space (regular plus invisible space) and "0" a single space.

2. For each group of 6 consecutive binary digits, convert that to base64.

Invisible

This is almost the same as spaces mode, but instead of using spaces plus an invisible character to encode 0s and 1s, we use only invisible characters. A 0 is encoded using Unicode character 00ad, and a 1 using character 200c. The result is a string that won't render visibly at all in HTML or text area, which PassLok places at the end of a dummy text line, with another dummy line following it. This way it is easy to edit the dummy text and make it something else with little risk of altering the invisible content. It is also harder to detect when selecting text. The invisible string can be copied and pasted to other areas, or sent by email, without affecting its integrity.

Of course, the Unicode characters that make up the "invisible" text are not invisible at all to a program that might be trying to detect them, but it may fool a human inspector lacking the appropriate scanning tool.

Letters

This method is adapted from the Advanced Unicode Stego by Adrian Crenshaw, 2013, which can be found at http://www.irongeek.com/i.php?page=security/unicode-steganography-homoglyph-encode. This form of encoding begins the same way as Spaces encoding: the item to be encoded is first converted into a binary string, at 6 bits per base64 character, based on each character's ASCII value. But instead of encoding this string as single or double spaces, it replaces the regular spaces with alternative Unicode encodings for a space. It uses codes 2004-2009 plus 202f and 205f, in addition to the standard 0020 code. Thus every space can encode three bits of the binary string.

In addition to spaces, alternative encodings are used for a number of letters that look the same (homoglyphs) in the Latin and Cyrillic or Greek sections of the Unicode chart. Thus, the original capital "A" (Unicode 0041), can be replaced by the Greek "A" (Unicode 0391). When the Latin "A" is used, a "0" is encoded, and when the Greek "A" is used, this means a "1". Not all Latin letters have non-Latin homoglyphs, but many of them do: a, A, B, c, C, e, E, g, H, i, I, j, J, K, M, N, o, O, p, P, s, S, T, x, X, y, Y, Z. The result is a text that looks identical to the cover, but encodes another text in a fairly compact fashion.

As in Spaces encoding, the cover text is repeated if more space is needed, and it is truncated when the complete binary string has been encoded. Since the output may be in the middle of a sentence, a warning tells the user that the text should be completed. Completing with Latin characters and regular spaces does not alter the material encoded.

To decode, the program finds the non-Latin characters and non-standard spaces and records a 1 (a three-bit code, for spaces), or a 0 if the corresponding Latin character was used. Then the binary string is converted back to the original characters.

Letters encoding produces the shortest, least detectable encoding (and this is why it is the default method), but has the disadvantage that a number of online services change the non-Latin characters and nonstandard spaces into their standard Latin forms as soon as the text is pasted on, thus destroying the encoded information. The user is warned to make a test with each particular service before using this encoding method.

Sentences

This method is closest to the Words method, already described, but rather than converting each original character into a two-word sequence, it is converted into a complete sentence from the cover text, plus punctuation. Six different punctuation marks are used at the end of each sentence: ,.;:!? and the initial letter of the following sentence is capitalized accordingly. The sentence itself is chosen by the number of letters contained in it, including spaces. Since there are 64 different characters to encode and 6 different punctuation marks, it is enough to have 11 different sentence lengths in order to encode all possible combinations. Here's the encoding process:

1. First take the cover text, split it into sentences, and measure the number of characters in each one. Then place each sentence in one of 11 arrays of a matrix, according to the number of

characters modulo 11. If at the end of the process any of the arrays remains empty, stop and register an error.

2. Now, for each character in the string to be encoded:

    a. Look up its index in the key alphabet, and compute index11 = index mod 11, and index 6 = int(index/11)

    b. Pick a random sentence from the index11'th array made earlier, and add the index6'th punctuation mark at the end

3. Capitalize according to punctuation, and add random linefeeds after periods, question marks or exclamation marks (40% works well).

And to decode:

1. Strip linefeeds and break up the text into sentences, making two arrays: one with the words (and spaces), and one with the punctuation at the end.

2. For each set of sentence and punctuation:

    a. Measure the sentence length and compute index11 = length mod 11

    b. Look up the index of the punctuation on this list: ,.;:!? This becomes index6

    c. Calculate index = index6 * 11 + index11

    d. Record the index'th character in the key alphabet

The result is grammatically correct text of a length similar to that of the Spaces output, but the sentences do not follow one another.

Image steganography

PassLok includes another way to conceal its items, and this is within images. Before version 2.4, it used the steganography.js library, by Peter Eigenschink for hiding into png images, plus the jssteg-1.0.js library, by Owen Campbell-Moore, for hiding into jpeg images. With version 2.4, steganography.js was dropped and replaced with an extension of the custom algorithm used for jpeg images, which still uses jssteg-1.0.js for basic encoding and decoding. The custom algorithm is an enhancement of the F5 algorithm, by Andreas Westfeld, devised to make sure that the histogram of coefficients of the original jpeg cover image is hardly changed at all by the encoding process.

In order to provide non-detectability, the algorithm involves a password, which is used to seed a PRNG (the current JavaScript implementation uses Isaac, derived from RC4). The PRNG then produces a random permutation of the image data to be used, which can later be reversed. The data is different for png and jpeg encoding. When encoding into a png image, the data is the pixel values for the red, green,

and blue channels, excluding the alpha channel because typically this channel does not contain any information (100% opacity for all pixels), and so anything encoded into it would be easily detected. Pixel data from areas not having full opacity are also ignored since these areas tend to have full black or full white pixel values and encoding there would be detected. When the output is a jpeg image, the data to be modified is the non-zero coefficient values for all the blocks that the jpeg image is divided into (zero coefficients are not included because then the image would fail to compress adequately, plus visual artifacts would appear in areas of solid color).

The first step, therefore, is to extract the data image to be modified by the encoding and put it all into a linear array. Then the PRNG generates a permutation of that array, based on the given password, thus scrambling the data to pseudo-random locations. The WiseHash key-stretching algorithm is used so that low-entropy passwords are stretched more than high-entropy ones. If no password is given, the PRNG is seeded with a string based on the image pixel dimensions and type. Since the password must be known both by the sender and the receiver, PassLok proposes a default value based on the sender's private key and the (single) recipient's public key (or shared key, if that is what is currently selected), which users can edit before the encoding takes place.

After scrambling the image data, encoding of the data to be hidden takes place. PassLok assumes that the data to be encoded, which is the output of a cryptographic algorithm is already statistically random, so the PRNG is not invoked again at this point, as it is in F5. To this it appends an end-of-data code consisting of 24 zeroes followed by 24 ones. Matrix encoding is used, so that first the program determines the largest k factor that will allow all the message data to be encoded within the available space (leaving some space for a hidden message, more on this later). The value of k is encoded first of all, using the fist four bits available in the image data. Then the image data is split into blocks of size 2^k – 1, and a single change made within each block (its location within the block is what makes the difference), depending on whether the output is png or jpeg:

- For png output, the image data (pixel color value) is raised by one if originally even, or lowered by one if originally odd, thus changing its parity.
- For jpeg output, the image data (DFT coefficient values) is decreased in absolute value by one (decreased if positive, increased if negative), with some exceptions. If the original value is one, it changes to minus one, if originally minus one, it changes to one (parity is re-defined in jpeg encoding so it is reversed from the normal sense for negative values), in most situations, but sometimes a one will change to a two, and a minus one to a minus two. Likewise, a two will sometimes change to a three instead of a one, and a minus two into a minus three rather than a minus one. This special choice is triggered whenever the number of ones or minus ones falls or exceeds the original value. In order to prevent the coefficient histogram as a whole to slide to lower values, an increment in absolute value takes place at random, rather than the usual decrement. The probability for this to happen is determined before the encoding begins, based on the original histogram of coefficients. This strategy avoids creating any additional zeroes (as F5 does) and maintains the histogram of the DFT coefficients very close to the original, making it very hard to detect the presence of the encoding.

After the encoding has taken place, the modified image data is returned to its original order by reversing the permutation and then it is injected back into the image. The user is then instructed to right-click on the image to save it to the clipboard or a local file.

Decoding begins the same as encoding, and requires the same password in order to scramble the image data in the same way, or otherwise the process will fail. The value of k is encoded in the first four bits, and then the rest of the image data is split into blocks of 2^k-1 bits. Then the matrix decoding algorithm is invoked on every block of image data, essentially involving a short hash of each block, whose value is the embedded message data. Rather than check for the end-of-data code after each block is processed, the entire image data is processed first and then we look for the end-of-data code. The latter option is considerably faster when coded in JavaScript. If the end-of-data code is found, the values preceding it are outputted as recovered message, otherwise an error is triggered.

We mentioned earlier that some space is reserved for a second message. If the user selects to have a second message (by having a three-part password, formatted as first password, vertical bar '|', second password, vertical bar, second message), then the process repeats using the unused image data, which is scrambled again with a new permutation based on the second password, and embedding the second message, before the first scrambling is reversed. When decoding, the user tells PassLok that a second message may be present by formatting the password field this way: first password, vertical bar, second password. PassLok will then attempt to extract the second message (which also ends with the end-of-data code) after extracting the first, using the second password to scramble the image data after the point where the first end-of-data code is found. The process will fail if either the first or the second password is incorrect.

It is worth mentioning that the process fails in exactly the same way whether the password entered is incorrect (or one of them, if there are two), or there is no data embedded in the image. In both cases, the end-of-data code is not found, and the same message is triggered. This means that an enemy cannot even detect whether anything is encoded into the image, unless he/she/it has the correct password(s).

User interface

Cryptography programs are notoriously hard to use, and so we have tried to make PassLok as simple as humanly possible. These are the design criteria:

1. As few different screens as possible. Avoid switching screens.
2. As few buttons and boxes as possible. Eliminate opportunities for confusion.
3. Avoid jargon that is confusing to users.
4. Provide feedback at every step: before and after things happen.
5. Automate whatever doesn't really require user intervention.

Therefore, the current version of PassLok is designed from the ground up with a graphical user interface, which is extremely easy to do with HTML and CSS. Most page elements are static are switched on and off by changing their visibility property. The app has a tabbed interface with three tabs that can be accessed at all times. These are:

1. **Main**. Data and results go here, as well as most of the action buttons.
2. **Options**. A collection of checkboxes that would clutter up the Main tab, for choosing operational modes. A few buttons used only for changing settings.
3. **Help**. Extensive documentation on all functions, written in a how-to step-by-step style.

In earlier versions, there was a fourth "Directory" tab, where users entered public and symmetric keys as well as items to be stored. With the introduction of a "Rolodex" list on the Main tab, this tab was demoted to the rank of special dialog for editing this list, which we judged less confusing to users, and is invoked by a directory "Edit" button on the Main tab. There are a number of such input dialogs, the majority of which are distinguished by the presence of a shadowed background and a smaller frame size to convey the idea that they are not permanent parts of the interface:

1. Directory edit: already mentioned, called by an "Edit" button on the Main tab.
2. Key entry: the first screen users see when PassLok loads, for entering the private key.
3. Email entry: in case the email/token is deleted from memory and is needed.
4. Parts input: used by the Secret Splitting function, asks how many parts to make and how many are to be needed to reconstruct the secret.
5. Decoy in: asks for the hidden message and the special key to encrypt it.
6. Decoy out: asks for the special key to decrypt the hidden message.
7. Key change: asks for a new Key twice, so everything stored will be re-encrypted with the new Key. Called by a button on the Options tab.
8. Email change: asks for a new email/token, which is then stored encrypted by the Key. Called by button on the Options tab.
9. Name change: asks for a new user name to replace the current one. Called by a button on the Options tab.

10. New User wizard: a collection of five successive screens that is called by a button on the Key entry dialog; each screen asks for a single input or offers a single output, all involved in setting up a private key.
11. Image input: to enter and display an image that is going to hide, or already hides, some PassLok output. Called from the Main tab.
12. General Directory: a frame for the separate General Directory web page. Called from the Edit dialog.
13. Chat: a frame for the separate Chat page. Called from the Main tab.

We now proceed to discuss features of some of these tabs and dialogs without getting into tedious programming details, which the reader can easily appreciate by looking at the code itself. Before we go ahead, we must mention that, in order to introduce users to PassLok as gently as possible, the program starts with a "Basic" interface comprising only the most essential buttons and settings. When they feel confident with the basic functions, users can switch to the "Advanced" interface comprising all the functions, by simply selecting a radio button on the Options tab. This choice is saved between sessions, so users always return to the last interface they used.

Main tab

The Main tab comprises two main elements, with accompanying buttons:

1. Local Directory: This is a dynamically-generated list of keys stored, listed by name given. The idea is that users only need to select the intended recipients from this list, and then clicking the Lock button will encrypt the contents precisely for these people to decrypt, without the users needing to handle the corresponding symmetric or public keys, which were entered once before. When the row of action buttons is shifted from cryptography to steganography functions, this list displays stored cover texts rather than keys. Lists of keys are identified by dashes at the start and end of their names, and are placed at the top. Other than this, the lists are alphabetical.
2. Main box: This is where plaintext is input and ciphertext outputs for encryption, and vice-versa for decryption. A single box seems the best way to prevent users from putting items in the wrong box. PassLok analyzes the contents and determines whether encryption or decryption is to be used. A single "Lock/Unlock" button that initiates both encryption and decryption, to prevent user confusion. A dedicated "myLock" button displays the user's Lock (public key) at any time. In the Advanced interface, a shift button resembling those found in calculators replaces the cryptography functions with steganography functions. Below the box are two rows of invariable buttons for displaying or hiding a formatting toolbar, clearing the box, or calling special functions to send email or initiate a chat session, plus buttons for loading and saving files.

The design principle of the Main tab is to allow users to do most of their work for a given session without leaving the tab.

Options tab

This tab contains a number of sections with selectors that would clutter up the Main tab if placed there, such as interface mode, encryption mode, text hiding mode, display modes, Short, Decoy, and Learn mode selectors, and buttons that are used only if the users wants to change settings. The settings shown depend on whether the user selected Basic or Advanced interface.

The settings change buttons deserve some further explanation. Items are stored into the localStorage variable proper of HTML5. This variable is a JSON object containing other objects, with this hierarchy:

1. Highest level keys: user names.
2. 2nd level keys, for each user name: named items on the local directory.
3. Arrays for each of those: shared or public key (or cover text) at index 0, other data at higher index, as described earlier.
4. One of these arrays, under key "myself", is special, and contains the user's public key (encrypted by the private key) plus a number of interface settings.

Since many of the stored items are encrypted by the secret Key of the user, PassLok asks the user to supply this Key before anything else is done. But what if the user writes the wrong Key (which by default is masked from view)? If this happens, there is a chance that items might be added to storage, encrypted with a (mistaken) Key that cannot be reproduced, leading to failure in the encryption of decryption processes later on. To prevent this, PassLok checks that the correct Key is entered by decrypting the encrypted salt value stored under key "myself". If it succeeds, the salt is used to compute the stretched private key that is seed for the signing private key and the DH private key, otherwise the program displays a message and asks again for the correct user Key.

But this makes it difficult for a user to change to a new secret Key, which should not be done lightly in any case because of all the stored items that are encrypted by the old Key. To facilitate things, PassLok decrypts with the old Key and re-encrypts with the new Key (entered twice so there is no mistake) every item in the local directory. This is what the Change Key button does. The Change Name button changes the string of the object key, at level 1 in the hierarchy, under which all the items are stored, and then re-encrypts the items as above because the key using for this encryption has changed as a consequence of changing the user name. The Change Email button only affects the one stored item containing this value, which is part of the 'myself' array.

The other two buttons on the Options tab are for backing up and possibly removing stored items. "Whole directory" takes the entire stored object at level 1 in the hierarchy, converts it to a string, and encrypts it with the current private key, then offers to delete it from storage. "Options only" involves only the "myself" array, which contains the encrypted public key and email/token, plus a series of interface items. It is normally encrypted, except if PassLok is operating in limited mode (explained later), in which case the items are not encrypted because no private key has been entered.

Help tab and other help features

A tab is dedicated for a document that provides step-by-step instructions on how to do things. Only the functions accessible in the current interface (Basic or Advanced) are displayed. There is also a search box to find keywords in titles. The structure of the document is a collection of headings which, when clicked, display the appropriate text. Another click on the title hides the content. In order to make the instructions as concise as possible, more detailed notes are kept in another collapsible area for some help items.

Contextual help is also given on the Main tab and the dialogs in several additional ways:

1. Message areas, usually located near the top of the tab or dialog, which give short instructions, confirm execution, and display errors when they occur.
2. Short strings explaining the function of each button appear when the mouse hovers over the button.
3. There is a Learn mode that can be set in Options. When this is set, a message explaining what is about to happen and seeking confirmation to continue will pop after a button is clicked.

Although it is possible to include step-by-step wizards for common functions such as encrypting and decrypting, we chose to have only one of such wizards, for the process of creating a new user. This is because new users presumably don't know anything about cryptography and need to be led by the hand. The new user wizard comprises five successive pages connected by navigational buttons:

1. Short description of what PassLok does and optional 3-minute video explaining the key concepts. The entire wizard can be skipped at this point via a button.
2. User name input, with explanation.
3. Secret Key input, with explanation and as-you-type strength measurement. There is a button to fill the box with five random words from the dictionary, if the user so chooses.
4. Email/token input, with explanation and button to generate random token.
5. Final button to display public key on Main tab, with further explanations. The wizard also offers to prepare an email to others containing this public key and instructions.

Guest mode

The normal mode of using PassLok is by entering the secret Key at the start, which enables the decryption of the stored items, but what if the user cannot or does not want to enter this Key at the start? There are a number of functions that do not require this Key or its derivatives, such as verifying a signature, encrypting in symmetric or Anonymous mode, decrypting a symmetric message, or joining/splitting a secret, plus all the steganography features. Should the user be completely locked out of these functions as well? The answer is guest mode.

A user enters guest mode by clicking the Cancel button rather than the OK button at the initial Key entry dialog. If a Key has been entered in the box, it is used, but PassLok still remains in guest mode. In this mode it is not possible to save anything between sessions, or use any stored item that has been

encrypted. Public keys are not stored encrypted by default, so they remain available unless the user has overridden this default via a setting on the Options tab. If a user Key has been entered (PassLok will ask for it if it is required), it is even possible to make signatures, encrypt in Signed mode, decrypt Anonymous and Signed messages, or display the public key matching that key. A record is made on the "myself" array that guest mode has been used in the previous session, so the regular user is made aware if someone not possessing the correct user Key has used PassLok since last time he/she used it.

Users are allowed to make one permanent change to storage while in guest mode, and this is to erase the "myself" array. Doing so allows writing in a different user Key that will be accepted, when PassLok reloads, thus restoring all functionality. The email/token string is also erased so it needs to be entered again. This should not be difficult unless this string was random, in which case the user won't be able to produce the same public key again. A warning to this effect is issued at the point where the stored email/token is erased, so the user can stop the process before it is too late.

Code structure and workflow

This last section is about how the code is organized, the different versions and their differences, and the process followed for improvements. Because of the need to authenticate the code in a way that is accessible to common users, the code served from websites contains almost everything in a single file, currently a little over 600 kB in size, plus a handful of css files containing alternate formatting information (no code allowed). The index.html file contains the following parts:

1. Heading and metadata, including a reference to an appcache so the code is cached locally and can run offline.
2. CSS section, where the default format for the different page elements is described.
3. JS code to force https rather than http
4. Open-source JS libraries, one after the other. Secrets.js must go first or otherwise there are errors.
5. JS code for PassLok, structured this way:
    a. Dictionary and blacklist. Long variable definitions, which are used by the key strength evaluation function and others.
    b. Initialization and Key and Lock functions. Everything that involves the user's secret Key, including the generation of the public key, and the Diffie-Hellman combination.
    c. Cryptography functions. Encryption and decryption for single and multiple recipients, followed by Digital Signature functions. Since encryption can occur in several modes, each function is divided into parts according to the different modes.
    d. Extra functions to generate formatted email, SMS, and chat invitations.
    e. Error correction functions. Reed-Solomon and best of three.
    f. Shamir Secret Splitting Scheme. This is barely a couple of functions, called from a single button on the interface.
    g. Steganography functions. First text-based steganography, then image-based.
    h. Local directory management. Everything about storing, finding, and modifying items stored between sessions through HTML 5's localStorage variable.
    i. Functions for interface management. What to do when screens open and close and buttons are clicked.

j. Chrome sync functions. These are only active for the Chrome app version.

6. Then the head closes and the body opens. First the interface HTML elements are formed, in this order:

    a. Tabs

    b. Main tab. This one has a complex structure:

        i. Two option lists to display the stored Locks and the cover texts, respectively. Only one of them is visible at the time.

        ii. First row of buttons, which again has three versions. One is displayed in Basic mode, the other two, one at a time depending on the state of one of the buttons, in Advanced mode.

        iii. Rich text formatting toolbar, including a number of graphic buttons described by dataURL rather than external files. Visibility of this toolbar is controlled by a button below.

        iv. Main box. Actually a div element, which is editable and can support rich formatting.

        v. Second row of buttons, including the one that toggles visibility of the formatting toolbar.

        vi. File input and output. One button to select the file, and one to save it.

    c. Options tab. This contains checkboxes and radio buttons for selecting the following:

        i. Basic vs. advanced interface. Depending on the selection, other sections below might be visible or not.

        ii. Encryption mode. In the basic interface, only Anonymous mode is allowed, and so this selection is not shown.

        iii. Other modes: Learn, Short, Decoy, etc.

        iv. Error correction checkboxes.

        v. Signing mode: Attached or Seal.

        vi. Steganography mode. Visible only in the advanced interface.

        vii. Buttons for changing the user name, Key, email/token (advanced only).

        viii. Buttons for backing up and optionally deleting the local directory or just the settings (advanced only).

d. Help tab. There are a number of help items, formatted that at the start only the heading is visible. A click on each heading displays the corresponding text below it. The text is usually formatted as a number list of steps. Many items contain a link to a video tutorial, which opens on a separate browser tab, and/or additional text that is displayed after a label is clicked. When basic interface is selected, only the items pertaining to the functions available in that interface are displayed. In the advanced interface all help items are visible. The help tab has a search box at the top, where headings can be found by keyword. The Privacy Policy and credits are located at the bottom of this tab, which can be reached by scrolling down.

e. Directory Edit dialog. This was originally a separate tab, which got downgraded in order to reduce the number of tabs. It contains buttons for saving, finding, editing, and deleting items stored in the local directory. There is also a button for accessing the General Directory, which loads into an iframe.

f. Intro wizard. This is divided into five screens, only one of which at most is visible at a given time. Buttons navigate between screens. Each screen contains an input box and a set of instructions to lead new users along the process of coming up with a secret Key and generating the corresponding public key (Lock).

g. Key dialog. This one contains the all-important Key input, which is the only place where the user's secret key resides. It is the first screen the user sees when PassLok starts, unless no Key has been used before (in his case the intro wizard is displayed). Since PassLok deletes the Key from this box after five minutes of not being used, the dialog is displayed again whenever the Key is needed again, after being deleted. An optional toggle suspends the Key-erasing timer.

h. Other dialogs called in special circumstances, such as in Decoy mode, the Shamir Secret Splitting Scheme, or when making a chat invitation. Invisible until called.

i. Whole screens to display an image used for steganography or for iframes containing the General Directory or the Chat session.

j. Finally there is a semi-transparent backdrop that gets placed behind the dialogs when those become visible.

7. After the interface elements are loaded, there are still two more pieces of JavaScript code, which would not run if placed in the head. The first contains interface initializations, window resizing, and file input/output functions. The second consists almost exclusively of event listeners attached to buttons and other page elements.

This is the way the single file is structured, when served from a web server. The code on GitHub is essentially identical, except that the JavaScript functions are placed in separate .js files rather than merged into the single file. This is done so the code is easier to audit. The Android version is based

directly in the GitHub code, which contains a few extra files used exclusively when compiling the Android version. The iOS version is based on the single-file web version. Finally, the Chrome app version is essentially the same as the GitHub version, except that the files involved in the Chat function are loaded locally rather than from PassLok.com, and are therefore included in an additional directory.

This is the typical development workflow of a new version of PassLok. The version number is of the form a.b.cc, where a is the major version, b minor version (public key compatibility), and cc the detailed version number:

1. Depending on perceived needs or bugs discovered, development may begin with the single-file, the GitHub version, or the Chrome app version. After a stable working condition has been achieved, a temporary file or commit is created, and changes are transmitted to the other two versions. Most commonly the work starts on the single-file version, with code segments collapsed for ease, normally using Dreamweaver as editor.

2. The next stage is to push changes to GitHub and generate and test the Android version. This is done automatically via Phonegap Build, which pulls the code directly from GitHub.

3. When no more changes are forced by Android compatibility, the single-file code is copied to a the location from where the iOS version is produced. Usually the only change required is to change the video URLs to "youtube:" statements. Then this version is compiled in Xcode and archived for distribution.

4. When everything seems stable, the single-file version is pushed to the servers, and the Android, iOS, and Chrome App versions are uploaded to the respective web stores.

5. The next step is to take the SHA256 (at least) and other checksums of the single-file code and publish it at several locations different from those serving the code: the three app stores mentioned above, the author's own website at prgomez.com, and the special PassLok information website at http://passlok.weebly.com.

6. The author also reads aloud the SHA256 hex code into a video that is posted on YouTube. The URL of this video accompanies the SHA256 value everywhere it is posted.