PassLok for Email technical document

(updated as of version 0.1, 4/17/16)

By F. Ruiz and team.

In this document, we explain what drives the design of the different components and functions of PassLok for Email, the version of PassLok that integrates with popular email services, 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. Much of the contents are copied from the "PassLok Technical Document," which does not focus on the email-integrated version of PassLok. This is a document in progress.

General design criteria

PassLok for Email is conceived as an easy-to-use, lightweight Chrome extension that adds some of the cryptographic capabilities of PassLok to popular browser-based email services like Gmail, Yahoo, and Outlook/Hotmail. The idea is to add high-strength encryption on a part-time basis to the user's preferred service, rather than to create a new service that users would need to connect to separately from their preferred email.

Here is a list of design goals:

- High cryptographic strength.
- Reasonable performance even on low-end computers.
- Inherently trustworthy. If the complete code is human-readable, so much better.
- No servers beyond the email provider. Therefore no liability from storing user data.
- Very easy to use even for cryptography laymen.
- Reduce key management to the minimum possible.
- Aimed at asynchronous communications, but can do synchronous as well if it is not too hard.

To meet these goals, PassLok for Email is a Chrome extension consisting essentially of JavaScript code, which has the following desirable traits:

- Great cross-platform compatibility, since the Chrome browser runs on any computer and any operating system with a decently modern browser available. Users also tend to be proficient with browsers, which enhances ease of use. Even though Chrome currently does not support extensions on mobile devices, there a plans for this in the near future.
- Auditability: the entire code is human-readable, so expert users are able to scan it for possible vulnerabilities without resorting to specialized software.
- Authenticity. Code delivered to individual machines is signed by Google so Chrome would refuse to install it if it has been tampered with. Additionally, it is fairly easy to

make sure that no changes have been introduced even without recourse to the signature, simply by comparing a one-way hash of the archived code with a trusted value.
- Computations coded in JavaScript are performed locally, without involving a server that might be compromised.
- Unlike regular web pages, Chrome extension code and data are protected against intrusion from web pages not in their manifest, and from all other extensions running in the browser.

There are also a few difficulties introduced with this approach:

- JavasScript code is not given full access to the computer hardware or the rest of the operating system. This is especially severe with Chrome extensions, which are not even able to open new tabs without an intermediate “background” page, or to pass data outside of the environment comprising their own code and data and those of the page with which they interact.
- 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 additional security measures involved in Chrome extensions actually create an environment that is safer from intrusion than the operating system itself.
- 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, 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, resulting from applying a key derivation funvtion to a user-supplied Password, 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 of PassLok Privacy (the standalone program from which PassLok for Email derives) 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 Privacy 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 Privacy 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 Privacy. 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 (both standalone and integrated with email) uses all of these primitives. We begin with symmetric encryption and decryption, which is at the base of all encryption modes. Symmetric encryption is also the type of encryption used in Invitation messages, where the recipients' public keys are not known at the time of encryption.

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 plaintext contains an encoded file, which is usually quite random-looking by itself and won't compress well, it is first compressed into a 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. 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. When encrypting an invitation, the sender's public key in base36 encoding is used as symmetric key. Since this is expected to appear random, its entropy is assumed to be maximum for its length, and so an exponent of value 1 is applied at the key-stretching step.
4. The stretched key and the compressed plaintext are then used as inputs to nacl.secretbox, which implements the XSalsa20 algorithm. 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.

   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 120 bits (20 base64 characters) are used for NaCl nonce.

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

1. The tags are stripped and the message is split into its constituents, in this order: sender's public key (if any), mode-identifying character, nonce, ciphertext. The modes are distinguished by the first character after the sender's 50-character public key, which is always a non-base64 character, according to this scheme:

Initial character(s)	Mode
@	Invitation (symmetric)
#	Signed (asymmetric)
$	Read-once mode 1 (asymmetric)
*	Read-once mode 2 (asymmetric)
~	directory item (symmetric)

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. Unless the result contains an encoded file (this can be easily detected), 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 for the user. Instead, it accepts as a valid Password, from which the private key is deterministically derived, whatever string the user wants to use as such. This is so that the user may remember the Password and not feel tempted to write it down, which might lead to its being compromised. Because many users tend to choose weak Passwords, PassLok applies a variable amount of SCRYPT key stretching, based on measured Password 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 Password, the program will run noticeably more slowly than with a strong one, because of the extra computations added by SCRYPT. This encourages users to choose strong Passwords that will be harder to guess.
2. All weak user Passwords 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 Passwords, 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 Password is first subjected to a variable number of rounds of SCRYPT stretching, as described above, resulting in a 32-byte stretched private 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 usually the user's email address, which can be extracted easily from the email program. In Outlook, however, the DOM does not contain users' email addresses but rather user names (addresses replace those in the server), and so those are used instead.

NaCl involves two kinds of private-public key pairs: on Curve25519 for performing Diffie-Hellman exchanges, or on the Ed25519 curve for digital signatures. PassLok Privacy, which includes signature functions in addition to encryption, uses the stretched user key as seed for an Ed25519 private key, and so PassLok for Email follows the same process in order to maintain compatibility with the standalone app. The key 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 Privacy parlance) 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 Privacy public key, also known as a Lock, looks like:

PL23lok==FWDpcOXA0AuW0SC/JLc2NMngivPM9zuDuY923UMqhkc==PL23lok

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.

In PassLok for Email (and also optionally in PassLok Privacy), users' public keys are displayed in base36 format rather than in base64, after a straightforward base conversion. This way, there is no confusion between capital and small case characters, thus facilitating the task of reading it aloud for authentication purposes. The base36 alphabet used in PassLok includes digits 0 to 9 and small case letters a to z, except for using capital L instead of small case l, which might be confused for digit 1. Thus, a 256-bit public key ends up being represented as a string of 50 base36 characters. This string is placed at the start of any output string, except data meant for storage.

Asymmetric encryption and decryption

PassLok for Email implements two main modes of asymmetric encryption, which are triggered by separate Encrypt buttons:

1. 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.
2. 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 some deniability (except for having been sent from a particular email account ;-), since it becomes impossible to detect who the sender or the recipient is, even if permanent private keys are compromised.

There is one additional variant of Signed mode. Chat invitations are especially encrypted messages which will automatically open a secure real-time chat session as soon as they are 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.

All modes admit multiple recipients whose identities remain unknown to one another (except, once again, for the addresses visible in the email page ;-). We proceed now to explain how the two modes of asymmetric encryption are put together.

Signed encryption/decryption

Signed mode encryption proceeds this way:

1. Unless it contains an encoded file, the message is compressed with LZ-String and encrypted with XSalsa20, using a 15-byte random nonce and a random 256-bit message key. Unlike in PassLok Privacy, no Decoy message and no padding are generated. 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 Password, followed by a ed2curve.convertSecretKey call when PassLok for Email first loaded, and is retained in memory.
    b. The recipient's Curve25519 public key is obtained from his/her stored 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: sharedSecret = nacl.box.before(publicKey,privateKey)

    d. Take the recipient's base64 Lock and encrypt it with XSalsa20, using the main message's nonce, and the recently calculated shared secret as encryption key. The first 9 characters of the resulting cipher text, encoded in base64, 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 strings obtained in this way: sender's Lock (public key) in base36 encoding, single "#" character, 15-byte nonce, then a "%" character, first recipient's ID tag, another "%", message key encrypted with the first recipient's shared secret, and then as many ID tags and encrypted message keys as additional recipients, all separated by "%" characters. Finally another "%" and the message cipher text.
4. The item is then bracketed by readable single-line tags so users can identify the kind of item this is. The body of the encrypted message is divided into lines of equal length and displayed.

Decryption is done this way:

1. Strip the initial and final tags and retrieve the initial sender's base36 Lock, then split the remainder of the string into its components. Those near the beginning are split at fixed positions within the string, and then the rest are split whenever a "%" character is found. The components are: mode indicator, nonce, then a series of ID tags and encrypted message keys in alternation, and finally the cipher text. The mode indicator should be the character "#", which means it is an asymmetric Signed message, which triggers the correct decryption algorithm without user intervention.
2. Now PassLok converts the sender's Lock back to base64 encoding and looks for it in the stored database, which is classified by sender's email. If it is absent of different from what was previously stored, PassLok stores the new string after asking the user to accept the change. Thus key exchange has taken place with a minimum of inconvenience.
3. The app now finds 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 base64 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 9 characters of the resulting cipher text constitute the "ID tag" for this message.

4. Search for the ID tag among the components of the message. If no exact match is found, this could have been due to the user having changed his/her Password since the last communication with the sender, so PassLok displays a dialog requesting the previous Password. If the ID tag derived from this one cannot be found either, PassLok displays a warning saying "there is no message for you" and stops. If there is a match, the component immediately following it is the encrypted message key.
5. 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.
6. 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 if it does not contain 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 features 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 as far as the message itself goes. 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. Now, the identities of the correspondents are likely revealed on the email metadata, so all this effort seems rather useless in PassLok for Email. It is retained, however, in order to maintain compatibility with PassLok Privacy, whose output is meant to be posted in public forums without leaking recipients' identities.
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

(other than from the email metadata) 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. Again, this seems spurious for emails, but it is retained for compatibility with PassLok Privacy.

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 (not used by PassLok for Email)

PassLok for Email does not have an Anonymous encryption mode, but PassLok Privacy does. Since its construction seems to understand Read-once mode, it is explained here even though it is not present in the app. 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, encoded in base64, is attached to the outgoing message immediately after the padding.

Upon decryption, once Anonymous mode is detected by the presence of the "!" character immediately behind the initial tag, PassLok knows that an ephemeral public key follows, 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 ephemeral 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's email address as salt) before they are stored:

Index	**contents**
0	Bob's public key (unencrypted)
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,A1),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 ("$" for Read-once and "*" for Read-

once) and performs the decryption accordingly. Other than this, the structure of the encrypted item is the same: initial tag, mode indicator, 20-character random nonce, 100-character padding (which may be random or contain a hidden message), alternating ID tags and encrypted ephemeral public keys plus encrypted message keys plus single-character mode indicator, separated by "%" characters (ID tags and ephemeral keys, plus message keys encrypted as described above), ciphertext (symmetric-encrypted by message key), hexadecimal error-correction code (if any), final tag. The mode indicator is "p" for mode 1, "o" for mode 2, and "r" for reset messages (more on this later).

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.

Still, it is possible for the exchange to go out of sync, so that a received message cannot be decrypted. One of the ways to cause this, for instance, is for Alice to send 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.

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 "r" 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

When the Read-once mode in PassLok was designed, the author was unaware that the Off-the-Record protocol (OTR) uses a very similar trick in order to achieve forward secrecy. OTR is closest to Read-once mode, 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 takes the same ephemeral private key that was used in that previous message, and sends out the same ephemeral public key along with it. OTR will keep reusing those values until a reply from Bob is received, at which point it will again produce a fresh set of ephemeral keys. Bob will be able to decrypt the first message he receives (which is not necessarily the first one that was sent), and if he receives more messages from Alice before he is able to reply, he knows they are encrypted with the same shared key, so he is able to decrypt them all.

But 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.

Key Management

One of the hardest problems in public key cryptography is how to generate, distribute, collect, access, use, and verify the private and public keys that are at the base of the whole cryptosystem, especially when the users themselves must be involved in the process. Most people do not have a clear understanding of the most basic notions, such as using the recipient's public key to encrypt rather than one's own. Confusion prevails, mistakes abound, and frustration sets in after a short time.

While many cryptosystems currently in use or under development try to solve this problem by setting up servers that collect, store, and distribute such keys, PassLok takes a very different approach and does away with special servers altogether. This is based on our experience with PassLok Privacy's own key server, named "PassLok General Directory". This directory, which can still be accessed at https://passlok.com/lockdir, is a separate web page loaded as an iframe within PassLok Privacy. Users voluntarily post their "Locks" (public keys), as generated by PassLok, so they are stored and classified under their email addresses and so other users can retrieve them. The General Directory requires email confirmation to post, modify, or remove a Lock, and to keep it listed after six months have elapsed. It has a simple but comprehensive help system and has worked without a hitch for two years so far (April 2016).

But it still has very few Locks listed. Many fewer than active PassLok users, if we count the statistics provided by all the sources of PassLok Privacy. This is possibly because users send their Locks directly to their friends, or attach them to their email signatures, or simply because they have no interest in communicating securely with people with whom they have no previous contact.

Therefore, PassLok for Email is designed so its public keys (identical to PassLok Privacy's Locks, for compatibility) are distributed at the same time as the messages are distributed, which is most easily achieved by simply attaching them to the messages themselves. Thus, the sender's Lock, consisting of 50 base36 characters (digits and small case letters, except for capital "L"), is prepended to every encrypted message. PassLok detaches this string and processes it before the actual decryption takes place, this way:

1. The Lock (public key) is converted to base64.
2. The sender's email address is looked up in the local directory (this was described earlier in the document).
3. If the entry exists, retrieve the stored Lock (1st item in an array) and compare it with the Lock attached to the message.
4. If the two Locks match, proceed to the decryption using that Lock. If they don't or the entry does not exist, display a dialog alerting the user of this fact and asking for

confirmation. After permission is granted, proceed with decryption using the Lock attached to the message.

5. In the second case, if decryption succeeds store the new Lock, otherwise keep the old one or leave the entry empty.

This way, users acquire other users' public keys (Locks) and keep them in a searchable format with hardly any involvement on their part. Locks needed for encryption are obtained automatically from storage, since they are indexed by email address. In order to maintain continuity as the user moves to other computers, the directory is not stored locally, but rather stored in "Chrome sync", which sends them to Google servers so the entire database can be retrieved as soon as the user loads PassLok under his/her Chrome account on another machine. Locks are not secret by nature, and so they are stored unencrypted. The ephemeral data that needs to be stored for Read-once mode is secret, however, and so PassLok encrypts it with XSalsa20 before storage, using the user Password stretched by WiseHash, with the user email as salt.

The process above works fine if one is always replying to someone else's prior message, since in this way the recipient's Lock will be stored before the reply message is made. If one is initiating the conversation, however, it is quite possible that a recipient's Lock is not yet known. PassLok handles this situation by means of special Invitation messages. This way:

1. If the Locks pertaining to some of the recipients are known but other Locks are unknown, PassLok encrypts the message for the known Locks and, rather than closing the encrypt window right after encryption, it leaves it open so the sender can see a message telling him/her which recipients won't be able to decrypt the message. Then he/she can close that window and remove those addresses from the recipients' list before actually sending the message.
2. If all of the recipients' Locks are unknown, PassLok hides the Encrypt button and its options and instead displays an Invite button, plus a special message telling the sender that the recipients need to be invited and that the invitation message won't be secure. The sender must click the Invite button twice (another message warns him/her of the lack of security), and then PassLok encrypts the message written in the box with the symmetric XSalsa20 algorithm only, using the sender's own Lock, which is later prepended to the encrypted message, as encryption key. This way the recipients retrieve the Lock and add it to their respective directories so they can reply to the sender with a normal encrypted message.

Decrypting an invitation message (type indicator character "@", plus special headers) triggers the display of additional instructions, which are added to the decrypted message and tell the user, who presumably is new to PassLok, how to reply to the sender. One may ask, however, why add an encrypted message at all, if it cannot be securely encrypted against eavesdroppers? The reason is this: having something to decrypt as part of an invitation message provides a further enticement for the recipient of the invitation to actually install the Chrome extension

and go back to the invitation email. If this decrypts successfully, and there is no reason why it should not be if the installation has completed since the decryption key is attached to the ciphertext, the user is rewarded, which further entices him/her to make an encrypted reply and thus establish full communication with the person who sent the invitation. In this way, public keys have been exchanged, stored, and made available, and the process was completed without the users having to this consciously. In their minds, they were just replying to an invitation someone sent.

Other functions

Decoy mode

Decoy mode supplements the regular encryption modes by encrypting a hidden message, which is then placed as the "padding" string of an encrypted item.

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.

Hidden messages are limited to 59 ASCII characters in length, 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 Anonymous 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, one's permanent private key 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 subject to Decoy decryption, by clicking a Decoy button on the decrypt window. Since the encrypted hidden message is meant to be indistinguishable from a random string, the recipient must tell PassLok whether the special key supplied is a symmetric key or a private key, which is done via a checkbox on the screen requesting special key input (default is symmetric key). If symmetric, PassLok simply uses it with nacl.secretbox.open and the same nonce as the main message in order to decrypt the hidden message. If private, PassLok combines it with the sender's public key, which is

included with the main message. The resulting shared secret is then used with nacl.secretbox.open and the same parameters as for a symmetric key.

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.

Real-time chat

PassLok Privacy 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 a new tab, and pass the rest of the decrypted material as a hash value (on the URL itself, following a "#" character).
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 by simply reloading the chat tab, in case the webRTC connections to one of several participants fail. 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.

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 Privacy, none of which requires the recipient to possess the cover text that the sender used for encoding. PassLok for Email uses the Letters method exclusively:

- 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.

- 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.
- Sentences: Each character of the original is replaced by a sentence from the cover text. The sentences are for 12 different lengths and end with one of six different punctuation characters.

As a table:

Method	**Spaces**	**Punctuation**	**Grammar**
Letters	variable	original	original
Spaces	variable	original	original
Words	constant	random	random
Sentences	constant	encoded	correct

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 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.

As mentioned earlier, PassLok for Email implements the Letters encoding method right after encryption, if so set via a checkbox in the encrypt dialog. Decoding is automatic as soon as the presence of encoding is detected, and performed prior to decryption.

The Letters 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 7 bits per 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.

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 a short, hard to detect encoding (and this is why it is the default method in PassLok Privacy, and the one used in PassLok for Email), but has the disadvantage that online services may 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.

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 intercepts all communications between Alice and Bob, and poses as the rightful sender or recipient in each message. 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 (Google, in the case of PassLok for Email), 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 because it purportedly came from his email address (which is what most "secure encryption" products in today's market rely on), because an adversary could easily intercept the emails and change them en route. Neither can she tell that Bob's public key is authentic just by looking at it. 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 so that 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 very easy to read without ambiguity because they are expressed in base36, which does not distinguish between lowercase and capital letters.

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 adversary 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. 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 Privacy (the standalone app) 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. Unfortunately, the key exchange method used in PassLok for Email involves adding the sender's public key to every encrypted message. Since the public key is usually known to belong to a particular person, the sender does not get external deniability. The recipients, however, do get this feature at least as far as the encrypted message itself. On the other hand, getting an encrypted message by email is incriminating enough even if the message itself cannot be tracked to each individual recipient. The feature is retained for compatibility with PassLok Privacy.

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 for Email salts the user-supplied Password with the user's current email address, as reported by the email page. Most email services report this in the normal username@servername.something, but some services, such as outlook.com, report something else instead (the user's complete name in the case of outlook.com). It really doesn't matter so long as it is distinctive to the user. 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.

An additional effect of this is that users always have different public keys for different email accounts, even if the Password used in all accounts is the same. This helps to keep the data separate.

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 for Email combats this attack in several ways:

1. PassLok for Email contains no instructions that would imply a connection to a server, with one well-defined exception: ephemeral keys are synced through Google servers. This is done using specific instructions that are validated at both ends, using TLS connections at all times. All private items synced are individually encrypted with the user's secret Key, salted with the reported email address, before they are synced.
2. All strings that might be user-generated, such as just-decrypted plaintext, are passed through a filter that deletes HTML tags 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 for Email 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.
4. Since PassLok for Email is a Chrome Content Script, and therefore it runs in a sandboxed environment that other extensions do not have access to, even if they have access to the underlying email page. Additionally, any highly sensitive piece of data, such as the user's

Password, is retained only temporarily, and then only through JavaScript variables rather than DOM elements.

Tracking

PassLok for Email 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 contact external servers under some circumstances, such as establishing a chat connection, which involves contacting 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.

In any case, chatroom information is not enough to establish a connection. One must also have a 256-bit random password, which is different for each chat invitation and is never sent to any servers. The chat program refused to connect to anyone who does not supply it correctly.

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 sent to the program that will actually do the transmission and the user must still send it manually. 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 for Email code is served from a public servers administered by Google, Inc. 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 for Email is most vulnerable. In order to combat it, we are following this strategy: a SHA256 hash is taken of every new version of the genuine PassLok for Email code, when compressed in .crx format, and this is posted on *different* servers from those delivering the code. Users are encouraged to check the hash before using PassLok. An item on the Help page 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 for Email 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. The user only needs to find a copy of Chrome, perhaps installed in a computer he/she has never used before, and log into it, and the extension becomes available right away with all its data.

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 for Email has saved locally or will be downloaded from the cloud as soon as Chrome starts? 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 Password is *never stored* anywhere, except in RAM. This Password resides only as a global variable, after it has been stretched into a private key. PassLok keeps a special timer that tracks the last time when the private key was used, and deletes it from RAM after five minutes of inactivity.
2. PassLok for Email stores ephemeral keys associated with other users, also encrypted by the user's Password (stretched with his/her email address). An attacker who obtains those could guess the Password by brute force or dictionary attack, and then he/she/it would have the secret Password as well as the contents. 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, stored items are padded with spaces so they are at least 43 characters long, before encryption takes place.
3. In the event of a user deciding that a machine is no longer to be trusted, the user only needs to remove his/her login from Chrome. Instructions are provided for doing this.