Secure Quick Reliable Login

This third "Cryptography" document in the four-document series assumes that its reader has read and understood the concepts described in the previous two "SQRL Explained" and "SQRL Operation Details" documents. This document described SQRL's use of cryptographic primitives. It details the algorithms, keys, signatures, and secure identity preferences storage system.

Documentation Download Links:

SQRL Explained	https://www.grc.com/sqrl/sqrl_explained.pdf
SQRL Operating Details	https://www.grc.com/sqrl/sqrl_operating_details.pdf
SQRL Cryptography	https://www.grc.com/sqrl/sqrl_cryptography.pdf
SQRL On The Wire	https://www.grc.com/sqrl/sqrl_on_the_wire.pdf

Contents

List of Figures

LibSodium

All of SQRL's cryptographic primitives are available from a single, popular, widely available, open, security audited library known as "Sodium" or "LibSodium." https://download.libsodium.org/doc/

The Sodium library describes itself as…

> *A modern, easy-to-use software library for encryption, decryption, signatures, password hashing and more.*
>
> *It can be cross compilable and is a portable, installable, packageable fork of NaCl, with a compatible API, and an extended API to improve usability even further.*
>
> *Its goal is to provide all of the core operations needed to build higher-level cryptographic tools.*
>
> *Sodium is cross-platforms and cross-languages. It runs on a variety of compilers and operating systems, including Windows (with MinGW or Visual Studio, x86 and x86_64), iOS and Android. Javascript and Web Assembly versions are also available and are fully supported. Bindings for all common programming languages are available and well-supported.*
>
> *The design choices emphasize security and ease of use. But despite the emphasis on high security, primitives are faster across-the-board than most implementations.*

We strongly recommend using LibSodium for SQRL's cryptographic primitives where possible. SQRL's identity storage system uses the AES-GCM cipher for authenticated encryption. The LibSodium implementation of this algorithm relies upon widely available hardware support for AES acceleration. This support should be available on all required hardware platforms, but if it is not, GRC has written a freestanding implementation in C which is freely available to any developers: https://www.grc.com/sqrl/aes-gcm.zip

Processing User Input

SQRL uses standard UTF-8 representation to efficiently handle non-ASCII international characters which might appear in passwords. Unfortunately, there are multiple ways for some UNICODE characters to be represented which, if not "normalized" could interfere with cross-device compatibility. Therefore, SQRL normalizes all incoming user-provided UNICODE characters by applying "NFKC" normalization. JavaScript defaults to NFC which doesn't push for the compatibility that we need. So SQRL uses NFKC. If you are unfamiliar with UNICODE normalization, this webpage provides a useful introduction:

https://withblue.ink/2019/03/11/why-you-need-to-normalize-unicode-strings.html

Client-Side Crypto

The first sections of this document describe the use of cryptographic algorithms by SQRL clients. One of the distinctive aspects of the SQRL system is that almost all of SQRL's crypto is on the client side. (SQRL's servers only need to verify the signatures of SQRL clients and generate unpredictable "nuts.") This is significant, since relatively few SQRL clients need to be created to provide complete platform coverage, whereas many more SQRL servers will need to exist to provide SQRL support across all server platforms. Therefore, having complex clients and simple servers greatly minimizes the total work required for full implementation coverage.

Harvesting Entropy

Compared to many cryptographic systems, SQRL's need for entropy quantity is minimal. However, its need for entropy quality is extremely high. SQRL clients generate their users' master identity from 256 bits of entropy, obtained from the client's environment. Cryptographic history is littered with examples of programmers relying upon their platform's native entropy generation only to later learn that the platform's "random number generator" was fatally flawed, or could be subverted in a way that seriously compromised the security of the systems built upon it. Therefore, we **urge** all SQRL client authors not to take their platform's entropy source at face value. Take some time to understand what it does and from where and how it obtains randomness. Regardless of a platform's underlying source or sources of entropy, implementing a "belt and suspenders" entropy generation system using a robust technique we term "entropy harvesting" is simple and powerful and will mean that you never need to worry about it again... even if your platform's built-in system turns out to be fatally flawed.

> Entropy: A property or an object? For convenience, this document treats "entropy" as both a property and as a noun. We'll say that we need 256 bits of "entropy." We understand that it is more precise to use more words, but in this instance, we have chosen convenience.

The Entropy Harvester

The Entropy Harvester's concept is simple: We create an open-ended SHA256 hash function into which many different sources of unpredictable data are continuously added ("poured") in the background. When a bit of entropy is needed, a snapshot is made of the hash's current state and that snapshot is finalized to produce the SHA256 hash of everything added to the hash since the start. In this fashion the SHA256 function is continuously mixing unpredictable data from many different sources to produce, upon demand, a 256-bit instance of entropy with all of the well-proven bit diffusion properties of that well-designed digest function.

The Sodium library facilitates the construction of this entropy harvester by offering exactly the SHA256 sub-functions we need:

```
int crypto_auth_hmacsha256_init(crypto_auth_hmacsha256_state *state,
                                const unsigned char *key,
                                size_t keylen);
```

```
int crypto_auth_hmacsha256_update(crypto_auth_hmacsha256_state *state,
                                  const unsigned char *in,
                                  unsigned long long inlen);
```

```
int crypto_auth_hmacsha256_final(crypto_auth_hmacsha256_state *state,
                                 unsigned char *out);
```

For the functions above, "`state`" is a variable of type "`crypto_hash_sha256_state`" and "`out`" is a 32-byte (256-bit) output buffer.

In a typical implementation, the **`crypto_auth_hmacsha256_init`** function would be called once during application start-up and provided with a "state" variable buffer. A background thread would then begin periodically collecting multiple sources of entropy and "pouring it" into the hash pot by calling the **`crypto_auth_hmacsha256_update`** function repeatedly in the background. The user's mouse movements could be collected and added, network timing values, disk access timing, CPU performance

register values, the state of the processor's clock counter, the exact UNIX time, and many other sources of unpredictable data would be added into the constantly churning hash pot.

This subsystem would provide an interlocked "mutex" function (to prevent reentrancy collision) which would make a copy of the current "state" variable into a "temp", run the **`crypto_auth_hmacsha256_final`** function against the "temp" to obtain 256-bits of hash, then discard the temporary state, returning a call-provided buffer filled with high-quality entropy.

If feasible for the client's platform, this "state" and "temp" memory should be allocated in non-swappable RAM-locked protected memory to minimize a local attacker's opportunity to obtain a snapshot of its contents.

SQRL's Keying Hierarchy

The previous two SQRL documents provided simplified diagrams and explanations of SQRL's synthesis and management of its keying. They were appropriate for introducing some of SQRL's new concepts such as its Rescue Code and password recovery. But our purpose in this document is to provide sufficient detail for SQRL implementers to understand all aspects of the system. Therefore, SQRL's complete "key-flow" hierarchy is finally provided without simplification.

The diagram below shows the way various functional modules are connected to each other and to SQRL's storage system. Please refer to this diagram for the enumeration of SQRL's various relevant connections and operations:

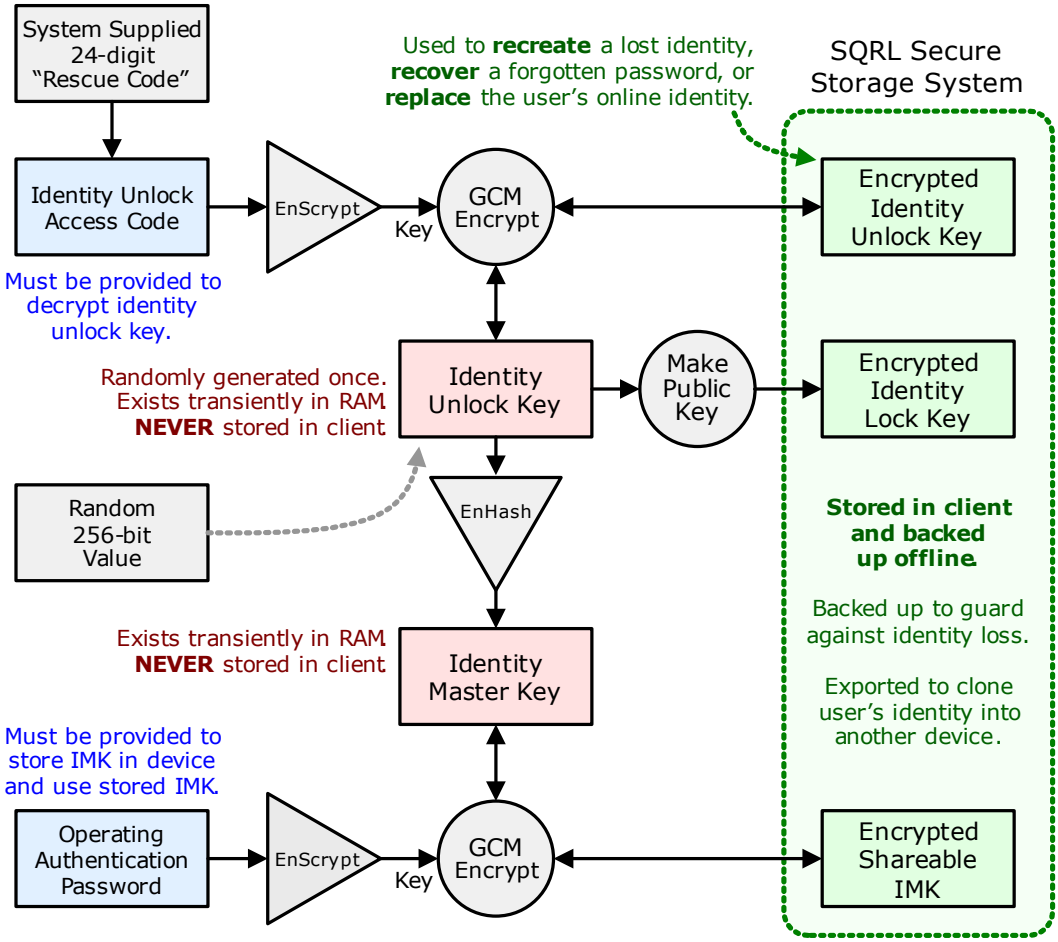

Figure 1: SQRL Key Flow Hierarchy

Identity Creation & IUK

Identities are created using SQRL's Entropy Harvester or guaranteed high-quality source of entropy. It enters the key flow at the middle left above as "Random 256-bit Value" to become the "Identity Unlock Key" (IUK). This can be regarded as the user's SQRL identity "grand master" key. It is **never** used directly. It is **never** written in non-encrypted form to any non-volatile storage or printout. As shown in the diagram above, the IUK has three connections to other components of the SQRL key flow.

Moving downward, SQRL's EnHash function is applied to the IUK to produce the IMK – the Identity Master Key. EnHash is used because we need an even more robust one-way guarantee than SHA256, when used in this limited input digest mode, would provide. So SHA256 is iterated 16 times with each successive output XORed to form a 1's complement sum to produce the final result:

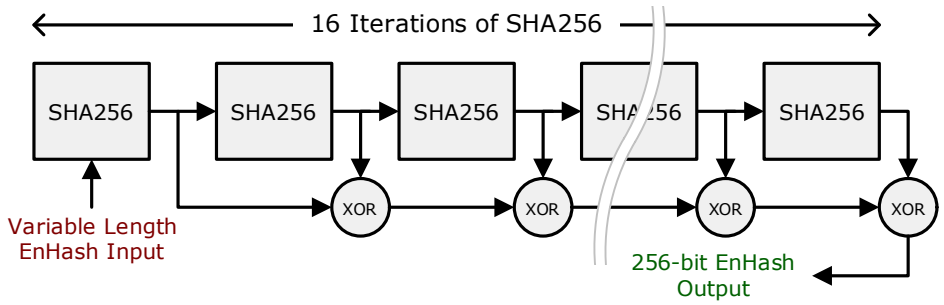

Figure 2: EnHash

Identity Master Key (IMK)

As the diagram shows, the output of the EnHash is the Identity Master Key (IMK). This is the key which is used to key the HMAC256 which hashes DNS domain names to produce the per-site public/private elliptic curve key pairs which represent and authenticate the user's identity.

Like its predecessor the Identity Unlock Key (IUK), this Identity Master Key (IMK) is **never** stored, displayed, or in any way output unless and until it has been encrypted. When it has been encrypted it is stored in SQRL's Secure Storage System format for safe keeping, export and inter-client transfer.

When the IMK is needed for use, it is loaded from the client's currently selected identity and then transiently decrypted into RAM for use. Both the encryption and decryption of the IMK are performed using a key derived from the user-selected and provided password:

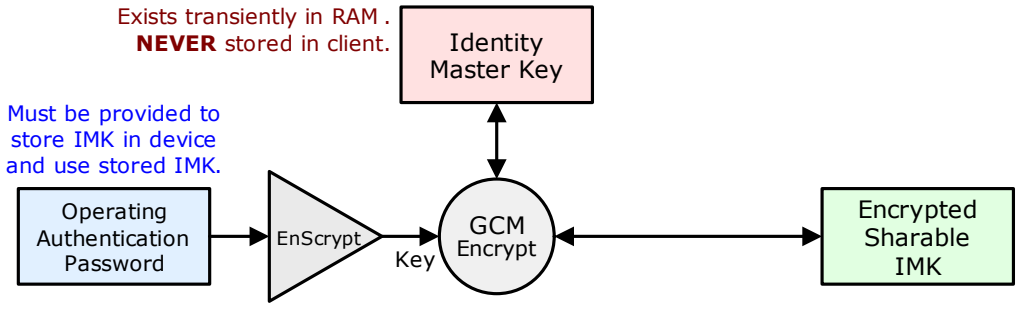

Figure 3: Encrypt/Decrypt IMK

When the identity is created, the randomly-chosen IUK is EnHashed to produce the descendent IMK. The user provides their own "access password" to encrypt the IMK and that password will henceforth be required to decrypt and use the stored IMK.

The User's Password

Because users may choose poor passwords, and because the decryption of the IMK for each use is not a time-critical event (as it would be, for example, if it was occurring on a busy server) SQRL uses the "Scrypt" memory hard function which, in SQRL's usage, requires a committed block of 16 megabytes of RAM. This moves the function safely out of the range of GPU's, FPGA's and ASIC's. However, even with this much RAM and using Scrypt's slowest parameters, Scrypt is still too fast. So SQRL has adopted a time-indexed solution which iterates the Scrypt function until a system- or user-specified amount of time has elapsed:

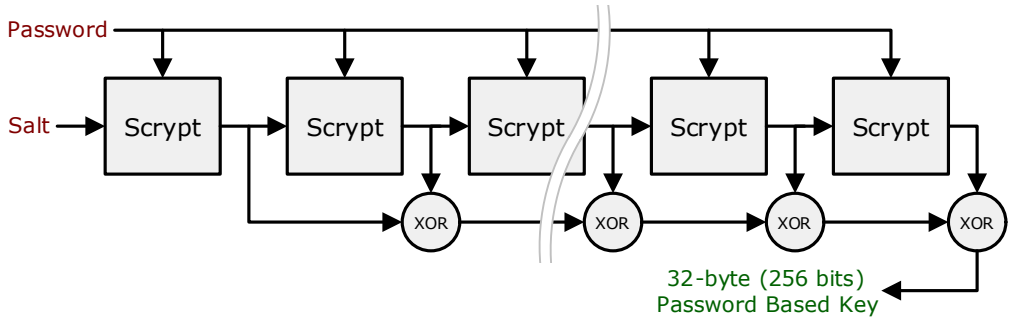

Figure 4: EnScrypt

The Scrypt function is governed by three parameters which have been carefully selected for maximum compatibility across all clients, since all SQRL clients must use exactly these Scrypt parameters:

N=512, R=256, P=1

Figure 5: Scrypt Parameters

Note that the "N" parameter is tunable to future-proof the system. It has the default value of 512 and is specified and stored within the SQRL identity storage structure. SQRL stores and uses the $\log_2$ (base2) of "N". Therefore, 9 specifies N=512 for a storage usage of 16 megabytes. All clients should default to this value for compatibility, the value may be tweaked in the future if needed.

To produce a password based derived key (PBKDF), the EnScrypt function iterates as shown in the diagram above until the system- or user-specified time has elapsed. During encryption, the number of iterations required are counted and stored so that the same number of iterations can be used during decryption to yield the same result when given the same input password. The three "EnScryption" durations are:

SQRL EnScryption Durations:

Password	5 second default. User customizable
QuickPass	1 second
Rescue Code	5 seconds

Figure 6: EnScryption Durations

The result of "EnScrypting" in this manner produces 256-bit symmetrical keys which are used with the AES-GCM function to encrypt and decrypt SQRL's various secrets.

The result of EnScrypting the user-provided password produces the symmetric key used to store the IMK after initial identity creation, and to later decrypt its stored form whenever it is needed to perform a SQRL authentication.

When the user has enabled SQRL's QuickPass feature, the first 'n' characters (default: 4) of the user's successfullyprovided password are EnScrypted for one second to produce a symmetric key which is used to encrypt the user's IMK for subsequent re-use by providing only those first 'n' characters. Note that the QuickPass material (the re-encrypted key, the AES-GCM random initialization vector (IV) and the iteration count and GCM authentication tag) are all retained in volatile RAM. Any QuickPass decryption failure immediately zeroes the stored QuickPass information and prompts the user for their full password.

The result of EnScrypting the Rescue Code and five seconds produces a symmetric key which is used to encrypt the master Identity Unlock Key and storage alongside the IMK in the user's identity store.

The AES-GCM Cipher Mode

The AES-GCM cipher is an AEAD (Authenticated Encryption with Additional Data) mode provides authentication encryption and decryption. It is perfect for our use since our application requires non-encrypted user option data which is, nevertheless, protected from undetected modification. This cipher mode requires a unique nonce that does not need to be secret. During encryption, an authentication 'tag' is produced and stored along with the encrypted result. During decryption this tag is provided and verified by the decryption function. A decrypted result will be provided by AES-GCM only if the tag matches, otherwise the decryption will fail with an authentication failure result. SQRL uses the success or failure to verify that nothing has been tampered with, and to verify that the user has provided the proper password, QuickPass or Rescue Code, since only the proper provided data will result in a correct authentication tag result.

The Rescue Code

Returning to figure 1, the SQRL Key Flow, and moving upward this time from the central Identity Unlock Key (IUK) we see a structure closely mirroring the password-driven ciphering we have just discussed. However, in this instance, the input is a system supplied 24 decimal digit Rescue Code.

At identity creation time, the system obtains 32-bytes (256-bits) of entropy. It then performs successive 32-byte long division by 10 until it has extracted 24 division remainder bytes, each with a value of 0 to 9. Note that the means for deriving 24 bytes of decimal data from 256-bits of entropy needs to be carefully considered because other ad hoc approaches might result in non-uniform digit distributions. This would critically weaken a very important security property of SQRL identities.

The 24 bytes of decimal data are converted to ASCII by adding 0x30 (ASCII '0') to each byte, and the resulting 24-character string is shown to the user for them to record and EnScrypted for 5 seconds on the device to obtain the IUK encryption key for the AES-GCM cipher. The cipher is applied with another initialization vector (IV) and the IV, result, the Encryption count and the authentication tag are stored into the user's identity.

Figure 1 shows how, if the user should forget their password to decrypt the IMK for use with authentication, the IMK can be reconstructed from the stored encrypted IUK as long as the Rescue Code needed to decrypt the IUK is known. This allows SQRL to provide password recovery without relying upon any other entity.

The Identity Lock Key

The last feature of SQRL's key flow hierarchy is the Identity Lock Key (ILK). As shown in figure 1, it is derived directly by passing the IUK through a "Make Public Key" function. We will discuss that function and the purpose of the ILK in our discussion of SQRL's Identity Lock, below.

SQRL's Site-Specific Key

The most often repeated diagram is SQRL's synthesis of per-site keys:

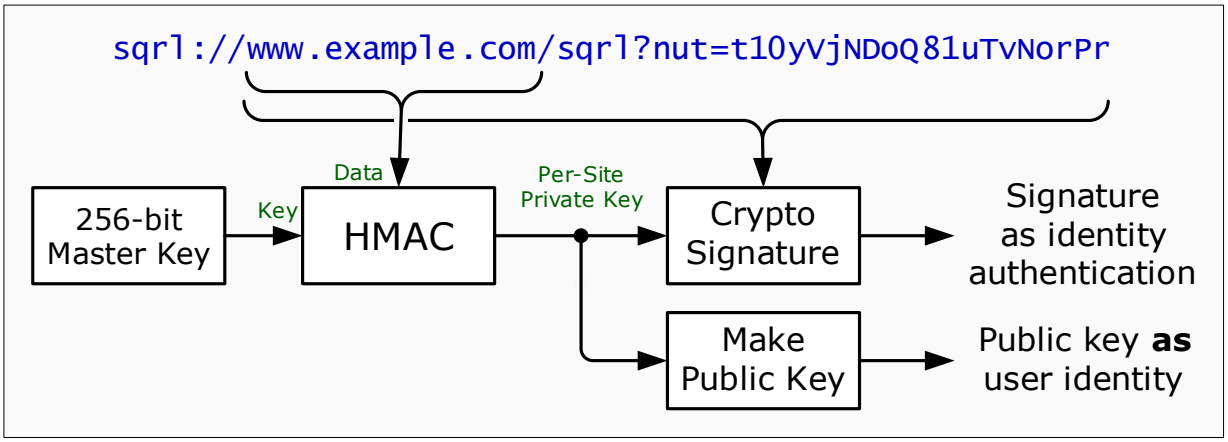

Figure 7: Per-Site Key Synthesis

We now know that the "256-bit Master Key" shown in this figure is the IMK described above.

The HMAC is a standard HMAC-SHA-256 and is offered by the Sodium library: https://libsodium.gitbook.io/doc/advanced/hmac-sha2:

```
int crypto_auth_hmacsha256(unsigned char *out,
                           const unsigned char *in,
                           unsigned long long inlen,
                           const unsigned char *k);
```

The 32-byte output of this **crypto_auth_hmacsha256** function is provided as the "seed" to Sodium's **crypto_sign_seed_keypair** function:

```
int crypto_sign_seed_keypair(unsigned char *pk, unsigned char *sk,
                             const unsigned char *seed);
```

... which returns the public key (pk) and the secret key (sk). The returned public key is the user's SQRL identity for the site whose domain name was hashed. The returned secret key (sk) is used by the **crypto_sign** function to produce the message signature(s) SQRL appends to each of its queries:

```
int crypto_sign(unsigned char *sm, unsigned long long *smlen_p,
                const unsigned char *m, unsigned long long mlen,
                const unsigned char *sk);
```

Note for those not using the NaCl-derived Sodium library, all of SQRL's signatures are based upon Ed25519 as described and detailed by Daniel Bernstein and colleagues. Information about it can be found here: https://ed25519.cr.yp.to/index.html. And Google will turn up many additional references and implementation of this elegant crypto signing system.

Secret Index and Indexed Secrets

There are applications where a web server might be storing data on behalf of its users which it does not want to be able to decrypt under any circumstances without the user's participation. In other words, the website doesn't want to have the key. SQRL supports this function with its "Secret Index" (SIN) and "Indexed Secret" (INS) parameters.

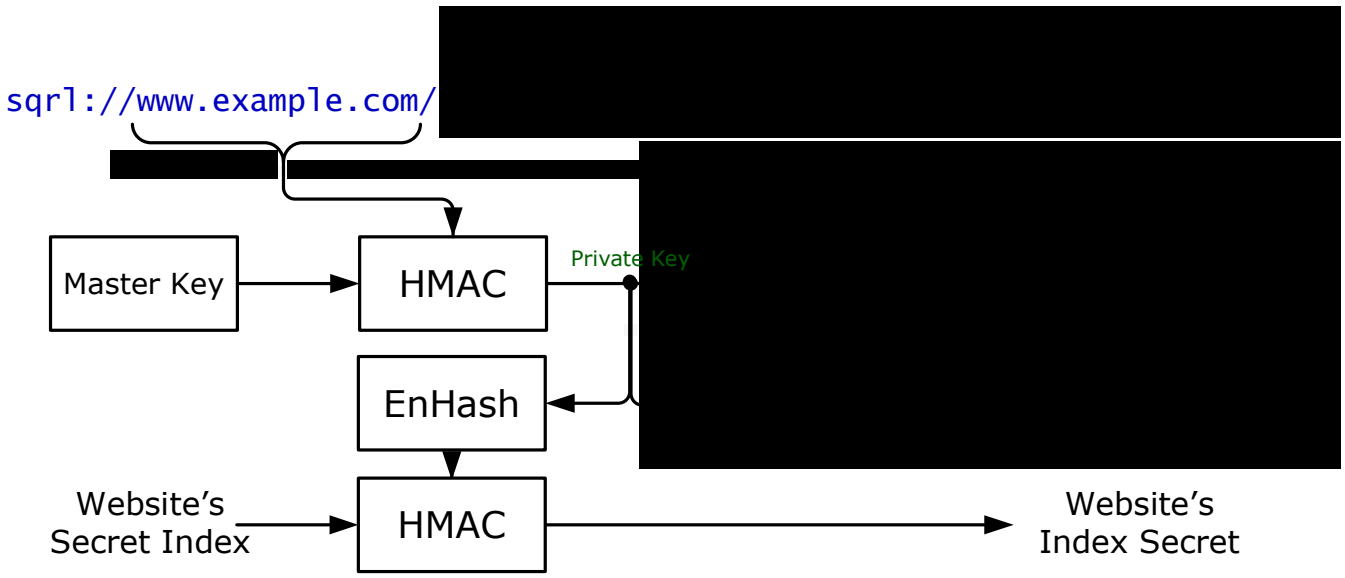

Figure 8: Forming Indexed Secrets

Any website wishing to take advantage of SQRL's Indexed Secret facility will return a "Secret Index" parameter in reply to all SQRL client queries. The secret index value can be anything the website wishes. As the diagram above shows, the user's per-site private key, which is synthesized from their identity and is passed through the EnHash function described on page 6 above. The output of the EnHash is used to key the HMAC-SHA-256 function which hashes the literal string that the website provided as the value for the "sin=" term. (See SQRL's 4th "On the Wire" document.) The resulting 256-bit binary key is base64url-encoded and returned to the website with the client's next query, if any.

Identity Lock

The user's view of SQRL's identity lock was presented in the first Explainer document. It provides SQRL with a number of desirable features by empowering SQRL's rarely-present Rescue Code with additional "powers". We've already seen how the Rescue Code has special power locally for password recovery. But the powers conferred by the Identity Lock operate across the Internet.

What the Identity Lock achieves

The Identity Lock protocol is a bit tricky. It needs to be more complex than SQRL's comparatively straightforward identification protocol because its requirements are more complex: It must be able to generate and provide something to every web server so that its identity can later be proven. But unlike SQRL's identification protocol, in the case of the Identity Lock, what it generates and provides to each web server cannot in any way identify its user to the server, since the SQRL system provides strong anonymity guarantees to prevent web servers from having anything they might compare to identify and track users. Additionally, since it must be immune to SQRL client compromise, the system that generates this identity-proving information must not, itself, be able to prove its own identity. Otherwise a compromised or hacked SQRL client could be used to maliciously prove its identity to

"unlock" and alter a user's website identification. So, we need to be able to provide anonymous identity proof without being able to prove our identity.

The Identity Lock is a cryptographic protocol which allows a SQRL client that is not carrying its Rescue Code (as SQRL clients never should) to ***establish*** new non-trackable assertions that can only be ***proven*** with the use of the identity's Rescue Code. In other words, SQRL clients can create a cryptographic assertion without the Rescue Code, that can only be proven when the Rescue Code is present.

This deliberate asymmetry enables SQRL clients to establish new SQRL identity associations that they are unable to change. This means that attackers who might obtain control of the client are also unable to change any previously created identity associations. Additionally, the Identity Lock is the key to SQRL's autonomous, yet secure, identity rekeying since the information required to rekey is available from the previous identity's Identity Unlock Key – thus the basis for its name.

Elliptic Curve Diffie-Hellman Key Agreement

In keeping with SQRL's use of Dan Bernstein's highly efficient and carefully designed Curve25519 elliptic curve cryptosystem, the SQRL Identity Lock protocol uses an elliptic curve version of Diffie-Hellman (DH) Key Agreement (KA). Therefore, our use would be known as Elliptic Curve Diffie-Hellman, or ECDH. For additional information see:

Dan Bernstein's ECDH web page: http://cr.yp.to/ecdh.html
and his Curve25519 PDF white paper: http://cr.yp.to/ecdh/curve25519-20060209.pdf

Traditional DHKA

In the traditional use of DHKA, two parties wish to establish a shared secret key which they will then use to encrypt subsequent communications. So, the parties first arrange to obtain each other's public keys. This can be done in full public view so long as no man-in-the-middle is able to intercept and subvert that public key exchange. In other words, a passive eavesdropper can be allowed or assumed to passively observe the exchange so long as they cannot alter it (to insert their own public key(s)). Then, with each party having their own matching secret key, each is able to compute the same unique shared secret by applying the DHKA process. This shared secret cannot be computed by anyone having or seeing only the exchanged public keys and, of course, each party has their own secret key which they never disclose.

If we have public key one (**PubKey1**) and secret key one (**SecKey1**) and public key two (**PubKey2**) and secret key two (**SecKey2**), then:

DHKA(PubKey1, SecKey2) = DHKA(PubKey2, SecKey1)

SQRL's use of DHKA

Some confusion can result from SQRL's unique application of Diffie-Hellman Key Agreement because it differs from the typical, more symmetrical use of DHKA. SQRL applies DHKA in a different, though valid, fashion. It uses three Diffie-Hellman properties that make it suitable for our needs:

1. **Secret keys are just random numbers.** It's extremely quick and convenient to be able to simply pick a large 256-bit number at random and use it as a secret EC key.
2. **Public EC keys have a hash-like one-way relationship with their matching secret key.** They are directly derived from their secret counterparts by running them through the EC function. This is a one-way function which, like a hash, does not reveal the secret key from the public key.
3. **Opposing pairs of keys combine to form a value that discloses nothing about either of them.** Owing to the one-way nature of EC crypto, the "agreed upon key" does not reveal either key to attackers. In practice, this characteristic is less than absolutely strong, so the key agreement output is typically run through additional hashing to strongly disconnect it from its source keys.

Each side of the equation above requires two keys, one public and one secret, from each of the parties. And here's the key: If one is missing, the proper value of that side of the equation cannot be calculated and the equality assertion between the two sides cannot be satisfied. The SQRL Identity Lock protocol relies upon that simple property.

Let's give these four keys their official names

One pair of keys is generated only once and never changes: The secret key is kept and stored offline for use only if the user's identity ever needs to be unlocked and changed. Therefore, we call this the "Identity Unlock Key." Its complimentary public key is retained online in the client and is used to provide identity locking information when the user first introduces themselves to a new website. It's called the "Identity Lock Key."

IdentityLock := MakePublic(IdentityUnlock)

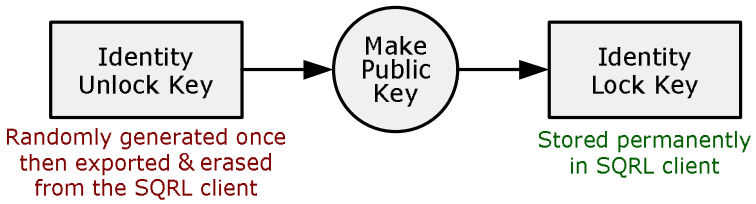

Figure 9: Identity Lock Key Synthesis

The second pair of keys is generated randomly every time a user is creating a new SQRL "identity association" with a website. First, a random 256-bit "Random Lock Key" is created. It is used to generate the identity locking information. Then, from it we generate the complimentary "Server Unlock Key" which the web server holds for us.

If we plug these four keys back into the original Diffie-Hellman Key Agreement equation, we get:

DHKA(IdentityLock, RandomLock) = DHKA(ServerUnlock, IdentityUnlock)

As shown in the two equations and diagram below, the system uses three of these four keys to synthesize the fourth Server Unlock Key, and an additional public Verify Unlock Key which are both sent to and retained by the web server:

VerifyUnlock := SignPublic(DHKA(IdentityLock, RandomLock))

ServerUnlock := MakePublic(RandomLock)

The "MakePublic" function appearing above uses the "`crypto_scalarmult_base`" function from LibSodium to create a Diffie-Hellman public key from a private key:

```
int crypto_scalarmult_base(unsigned char *q, const unsigned char *n);
```

The "SignPublic" function is the "crypto_sign_seed_keypair" function from LibSodium. It is part of the Curve25519 elliptic curve cryptosystem which converts a 256-bit "seed" into a pair of signing keys: a private signing key and a public signature verification key. It's the same function used for SQRL's primary identification keying mentioned earlier:

```
1  int crypto_sign_seed_keypair(unsigned char *pk, unsigned char *sk,
2                               const unsigned char *seed);
```

The VerifyUnlock key is the **public signature verification key** produced by that function.

After examining this description of key and signature definitions and functions, we'll step through the identity lock and unlock process:

Description of SQRL's Identity Lock Keys and Signatures

Identity Unlock Key: (IUK)	Randomly generated once, then printed and stored offline. The SQRL client must have access to this secret key for it to synthesize the Unlock Request Signing Key which the client must use to sign any identity change request.
Identity Lock Key: (ILK)	Generated once, but it can be regenerated if needed, because it is the public side of the Identity Unlock Key. It is retained by all of the user's SQRL clients and combined with randomly generated Random Lock Keys to create a web site's Verify Unlock Key (which verifies the signatures of any identity change requests).
Random Lock Key: (RLK)	Randomly chosen whenever the user is associating their SQRL identity with a new web server. It is used for two things, then discarded: It is mixed with the permanent Identity Lock Key (above) to produce the Verify Unlock Key, which the web server receives and stores. It is also used to create the Server Unlock Key which is also sent to the web server and stored.
Server Unlock Key: (SUK)	Generated by the SQRL client then sent to and retained by the web server. When the client later indicates that it wishes to change its identity association, the web server returns this public key to the client. The SQRL client combines it with its reloaded secret Identity Unlock Key to produce the Unlock Request Signing Key which must be used to sign any identity change request.
Verify Unlock Key: (VUK) (Signature)	Received and retained by the web server from the client during the initial SQRL identity association, it is the public key used to verify the signature produced by the Unlock Request Signing Key. The web server will use this key, as it does with the SQRL identification protocol, to verify that the SQRL client is in possession of the matching Unlock Request Signing Key (which it could have only synthesized by combining the web server provided Server Unlock Key and its own secret, reloaded, Identity Unlock Key).
Unlock Request Signing Key (URSK) (Signature):	Synthesized by the SQRL client by combining the web server supplied Server Unlock Key with the user's secret reloaded Identity Unlock Key. This produces a secret elliptic curve signing key that must be used to sign any SQRL client's identity change requests. The server's matching Verify Unlock Key is the public key used to verify the signature.

How SQRL Identities are Associated and Locked

The SQRL client continuously holds onto the Identity Lock Key after it is created from the Identity Unlock Key. We cannot simply give every web server the public Identity Lock Key, because doing that would mean that we were giving every web server the same public key . . . which would break one of the SQRL system's most important properties: No ability to link user identities between servers.

Instead, we create a unique randomly derived version of our Identity Lock Key that cannot be traced back to our actual Identity Lock Key. We do this by first generating a new Random Lock Key and use the DHKA function to mix it with our Identity Lock Key (see below). This produces a 256-bit value that

will be our private identity request signing key, but we don't use or retain it at this stage. Instead, we immediately pass it through Dan Bernstein's elliptic curve signing algorithm to produce its matching signature verifying public key. This becomes the Verify Unlock Key that is sent to and retained by the web server for it future use in verifying the signatures of the client's identity change requests.

In order for the client to later be able to re-synthesize the matching private signing key (which we just ignored and discarded), the Random Lock Key is also passed through a standard Curve25519 Diffie-Hellman Key Agreement (DHKA) function to create its matching public key. This becomes the Server Unlock Key which is sent to the web server for storage. When the SQRL client indicates that it wishes to update this user's identity, the web server returns this key which, when combined with the user's super-secret Identity Unlock Key allows the SQRL client to synthesize the private elliptic curve signing key which is required to sign any identity change requests:

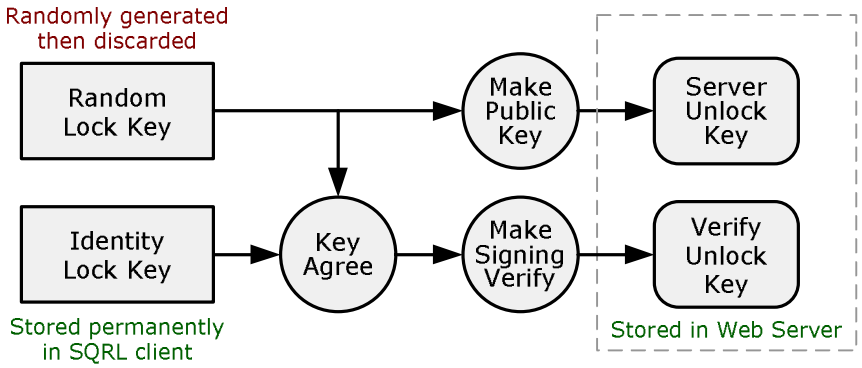

Figure 10: Creating a New Identity Lock Association

When a user is first associating their SQRL identity with web server account (which we refer to as "identity association") the SQRL client generates (see the figure above) and the server receives and permanently stores the Server Unlock Key and the Verify Unlock Key which, as shown in the diagram above, are created on-the-fly and derived from a random number and the user's permanent Identity Lock Key. Once those keys have been set for a user, they never need to be changed. They are both unique public keys which disclose nothing about the user's identity. They serve to lock the value of the user's SQRL public identity key, which was provided when the identity association was established.

How an Identity Association is Unlocked

Although this has already been touched on above, we'll examine it in greater detail here.

For an existing SQRL identity association to be re-enabled, updated, replaced or removed, the SQRL client must be able to cryptographically sign its identity change request by regenerating the correct private elliptic curve signing key. In response to the SQRL client's request to change identity association, the web server sends the SQRL client the stored value of the Server Unlock Key. This is the value the SQRL client originally generated and asked the web server to retain for just this eventuality. The Server Unlock Key provides the public half of a standard Diffie-Hellman Key Agreement (DHKA) where the private half is the user's super-secret, normally offline but now reloaded, Identity Unlock Key:

```
UnlockRequestSigning := SignPrivate( DHKA( ServerUnlock, IdentityUnlock ))
```

Similar to the VerifyUnlock key above, the "SignPrivate" function is the "crypto_sign_seed_keypair" function from LibSodium. It is part of the Curve25519 elliptic curve cryptosystem which converts a 256-bit "seed" into a pair of signing keys: a private signing key and a public signature verification key. The UnlockRequestSigning key is the **private signing key** produced by that function.

Since this is the right-hand side of the original equation above, it reproduces the private Unlock Request Signing Key which was briefly created but never used, other than to produce the matching public signature verifying Verify Unlock Key which the server also stored. The SQRL client combines

the web server provided Server Unlock Key with the user's reloaded Identity Unlock Key to reproduce the original Unlock Request Signing Key. The client uses this recreated signing key to cryptographically sign its identity change request and sends that specially signed request to the web server.

The web server uses its stored Verify Unlock Key, which is the matching public signature verification key to verify the request's validity and authenticity and, if verification succeeds, performs the requested highly privileged function, updating the appropriate entries in the user's SQRL account database.

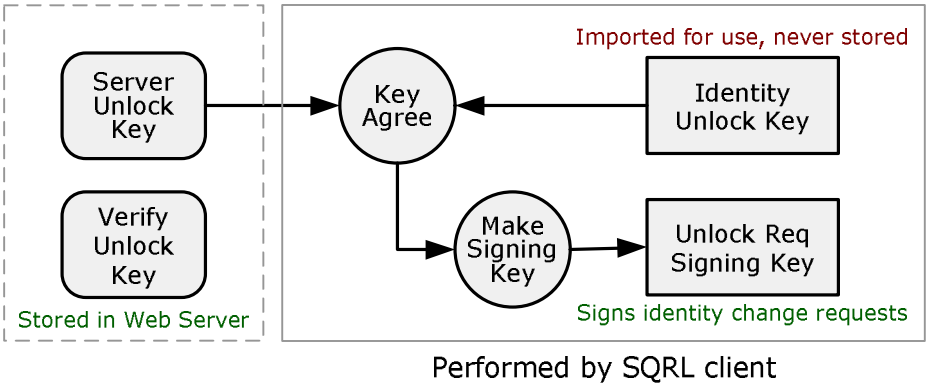

Figure 11: Unlocking the Identity Association

A Few Technical Identity Lock Notes:

- The web server provides a cryptographically strong "nut" in every SQRL link URL and with every SQRL client query reply to protect all transactions and signatures from replay attacks.
- Note that to protect against replay attacks the web server must verify that the nut value returned by the SQRL client exactly matches the nut it supplied. This prevents an attacker from reusing a previously captured nut to again succeed with a replay attack.
- Since a new 256-bit Random Lock Key is only needed when a new identity association is being established with a web server, SQRL's demands for platform entropy are already very minimal. Moreover, the identity lock protocol has very weak entropy requirements. The Random Lock Key is never exposed outside the client and it is situated behind one or two elliptic curve modular multiplications which are lossy one-way functions. So, unlike with SQRL's main operating secret keys—whose entropy requirements are extreme—there is little need to worry about or strengthen the generation of this transient and discarded key.
- It would have been more convenient to use the SQRL system's existing site-specific public and secret keys instead of continuously generating random keys for the Identity Unlock and Identity Lock keys. But if we had done that, an attacker who had compromised the user's password and identity master key would have been able to compute the value of the Server Unlock and Verify Unlock keys, thus defeating the system's protections. So the system uses random key values.
- For more information about the detailed operation of Diffie-Hellman key agreement, see the very good Wikipedia entries:

  https://en.wikipedia.org/wiki/Diffie%E2%80%93Hellman_key_exchange
  https://en.wikipedia.org/wiki/Elliptic-curve_Diffie%E2%80%93Hellman

Message Signatures

The fourth and final "On The Wire" document in this SQRL documentation series provides details of the SQRL client and server interchange. But we need to take a look ahead to discuss the cryptographic signing of SQRL's client messages.

SQRL clients communicate with SQRL servers using standard HTTP POST queries. The initial URL is obtained from the "sqrl://" URL after converting it into "https://". Each subsequent query URL (each containing a new unique nonce from the server) is obtained from the server's immediately previous reply to the previous query. Thus, SQRL client queries are conceptually chained together. No SQRL client is ever required to query the server, and the server's final SQRL URL will always go unused.

The body of every SQRL POST query, which contains its query parameters and a copy of the server's previous initial URL or previous reply, is signed by at least one and often by several cryptographic signatures. The server does not sign its replies, but every SQRL query returns the unmodified content of the server's previous reply which the server **must** arrange to authenticate to detect any modification to what it previously sent. In the event of any modification, the server would abort, ignore the query, and return an error the client.

Note that the server's reply authentication could be achieved by appending a secretly keyed HMAC to its reply which would be verified upon return. Or it might compute the SHA256 digest of its reply and retain it server-side for subsequent comparison with what the client returns.

After the client has assembled its query envelope consisting of its query parameters and the server's previous reply, one or more elliptic curve signatures are computed and appended to the envelope. The Sodium library's **crypto_sign_detached** function is used to produce the message signature(s) SQRL appends to each of its queries:

```
int crypto_sign_detached(unsigned char *sig, unsigned long long *siglen_p,
                         const unsigned char *m, unsigned long long mlen,
                         const unsigned char *sk);
```

Identity Storage

SQRL's S^4 (SQRL Secure Storage System) is a secure, tamperproof, extensible, encrypted, uniform structure which contains SQRL user options, the user's current Identity Master Key, Identity Lock Key and Identity Unlock Key. It may also contain zero or more previous Identity Unlock Keys.

This structure is not only used internally by all SQRL clients, but it is the format used for textual and QR code identity storage backup and inter-client identity exchange. It was designed for SQRL and offers a number of features which make it highly suitable for these applications.

Secure Storage?

"Secure Storage" in this context means that data can be optionally encrypted for privacy, but whether or not encrypted, any alteration of stored data can be reliably detected. In cryptographic terms, this provides authenticated encryption with associated data (AEAD).

SQRL clients require both offline and online storage security:

SQRL's need for secure storage includes long-term offline storage of the user's master identity and identity unlock keys. Any time these extremely sensitive keys are exported from a SQRL client — even when jumping between devices as a QR code — they are encrypted under a random key derived from SQRL's EnScrypt memory and time-consuming attack resistant PBKDF.

To enforce key privacy, SQRL uses strong encryption. And to verify successful decryption, verify the user's provided passphrase and detect any tampering, SQRL employs strong message authentication.

To provide additional security, SQRL clients always retain their users' internally stored keys in encrypted form. They are decrypting transiently on-the-fly only during the moment they are needed. Therefore, the plaintext form of SQRL's sensitive keys exist only transiently.

SQRL's Secure Storage System (S^4) incorporates the following features:

- An S^4 database is identified by its first eight-byte signature 'sqrldata' appearing in string-sequence with 's' being the file's first byte.

- Although S^4 is nominally a binary representation, it may be necessary to convey the file over a binary-hostile medium. In this case the standard eight-byte signature will be converted to all uppercase 'SQRLDATA' and the balance of the database will be encoded into 6-bit base64url format for representation, storage or transmission over a binary-intolerant channel.

- When a SQRL client encounters a database beginning with uppercase 'SQRLDATA' it must immediately lowercase the 8-byte header and pass the balance of the database through a base64url decoder to restore the database to binary. Processing may then proceed normally.

- To aid in the exchange of SQRL data during development — posting on forums, etc. — base64url-illegal line ending and whitespace characters — CR, LF, TAB and SPACE — should be silently ignored for line wrap tolerance.

- Strings are stored in their natural order, first byte first, and multibyte numerical values are stored in 'little endian' order, with their least significant byte first. This is the reverse of the so-called 'network byte order', but this is not a network protocol, it is a storage format and the order is not important as long as it is clearly specified. All Intel and AMD systems are all natively (pardon the pun) little endian, and ARM-based systems are configurable, with many mobile systems choosing little endian mode for inherent Intel compatibility.

- All numeric values are simple, unsigned, binary.

- Each block begins with a 2-byte length specifier which includes the length byte's two bytes. Similarly, lengths would typically fit within a single byte, but two bytes allows for ample future growth.

- The second datum in each block, following its length bytes, is a two-byte block type. As shown above and described below, only three block types are defined by the version 1 S^4 specification. So, again, plenty of accommodation for growth.

- This simple, lean, and efficient structure allows SQRL to use a minimal-size credential storage file format while supporting virtually any degree of future expansion in block size, block type, and block count **. . .** all while maintaining backward compatibility to the original lean v1 specification.

Backward compatibility and forward growth

All SQRL clients wishing to interchange SQRL identities must fully comply with this simple and straightforward specification. Though this specification is not complex, the following easily enforced characteristics provide significant future growth and backward compatibility:

- This specification fully describes three block types and any unknown block type should be ignored. This allows future clients to incorporate additional features and capabilities in a backward compatible fashion. Older clients, unaware of other block types, will simply ignore them.

- The currently defined layout of existing block types must be honored. SQRL clients wishing to store additional non-standard data in SQRL identities must define and create additional block types. S^4 clients will ignore any data they do not explicitly expect.

- Future clients may incorporate support for new PBKDF or Authenticated Encryption modes, thus augmenting or replacing the v1 use of EnScrypt, SHA256-Scrypt, AES-256-GCM, by defining new block types and providing whatever data they require.

- The type 2 and 3 blocks use an implied NULL initialization vector (IV) nonce. This is cryptographically safe because the application of the type 2 and 3 blocks precludes the same data ever being encrypted by the same key. Since this is not true for the type 1 block, its plaintext header region does incorporate an initialization vector (IV) nonce.

- All blocks are terminated by a 16-byte verification tag. This authentication tag is generated by the AES-GCM authenticated encryption and must always be the last field of any block.

AES-GCM: "AES/Rijndael - Galois/Counter Mode"

SQRL obtains both robust data encryption and decryption verification (wrong password, error and tamper detection) through its use of the industry standard AES-GCM "authenticated encryption" (AE). Unlike earlier authenticated encryption schemes which either applied a MAC (message authentication code) to the encrypted data, or applied encryption to data that already had a MAC authentication, the AES-GCM performs both operations at once and does not require the use of separate keys for each aspect of the operation.

AES-GCM has been adopted by NIST (National Institute of Standards and Technology). It is present in the IPsec (IP Security) specification and in the TLSv1.2 security suites.

```
int crypto_aead_aes256gcm_encrypt(unsigned char *c,
                                  unsigned long long *clen_p,
                                  const unsigned char *m,
                                  unsigned long long mlen,
                                  const unsigned char *ad,
                                  unsigned long long adlen,
                                  const unsigned char *nsec,
                                  const unsigned char *npub,
                                  const unsigned char *k);
```

```
int crypto_aead_aes256gcm_decrypt(unsigned char *m,
                                  unsigned long long *mlen_p,
                                  unsigned char *nsec,
                                  const unsigned char *c,
                                  unsigned long long clen,
                                  const unsigned char *ad,
                                  unsigned long long adlen,
                                  const unsigned char *npub,
                                  const unsigned char *k);
```

As shown above, the AES-GCM encryption and decryption ciphers are available in the Sodium library, though they do make use of some specific hardware instructions which have not always been present:

https://libsodium.gitbook.io/doc/secret-key_cryptography/aead/aes-256-gcm

In case the Sodium library's hardware dependence is a problem, GRC has produced and provides a reference implementation of AES-GCM whose source code is released without any usage restrictions of any kind into the public domain. It was written in platform-neutral 'C' and should be easily compiled for use with SQRL. Included in the source distribution are NIST's six files of algorithm validation test vectors, a PERL utility to compile them into a single binary, and a test harness (gcmtest.c) which validates GRC's implementation against NIST's 47,250 individual test vectors.

https://www.grc.com/sqrl/files/aes-gcm.zip

Efficient, compact, binary-friendly, future-proof SQRL client storage

The trend, exemplified by XML and JSON data structure description languages, reduces information storage density and efficiency in favor of embedding structural descriptive flexibility. Bundling an application's data description metadata into the data can simplify dissimilar system inter-operation by allowing the data to describe itself. And in today's world of terabyte storage and gigabit communications, the cost of embedding inter-operation metadata is offset by interconnection robustness.

But an excess of metadata makes little sense for SQRL's offline static data storage where long term storage robustness and retrievability are the overriding priority. SQRL exports long-term statically encrypted secrets in future-proof printed paper form, so fewer bits means that each bit can be larger. Fewer characters means that each character can be larger. Few bits means there's more room for stronger error correction with its larger overhead, etc.

The other problem with metadata-laden representations is that their serialization and deserialization can become a significant source of security vulnerability. Many past exploits have taken advantage of the too-casual interpretation of stored data when it needs to be imported.

However, any storage format defined today must provide for future change and expansion. SQRL's use of first-byte tagging for block type allows for significant expansion in the future.

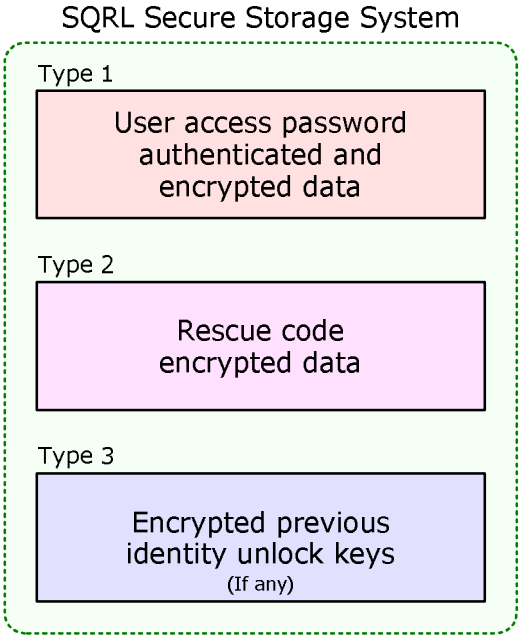

Figure 12: Secure Storage Layout

Tagged Blocks

As shown above, SQRL's minimal standard identity storage is divided into three blocks each containing a subset of the user's identity data encrypted under a different key.

- Type 1: Password based PBKDF key, plaintext data and encrypted identity keys.
- Type 2: Rescue Code based PBKDF key, encrypted identity unlock key.
- Type 3: Encrypted previous identity unlock keys.

Type 1 - User access password authenticated & encrypted data
The type 1 S[4] block supplies the EnScrypt parameters to convert a user-supplied "local access passcode" into a 256-bit symmetric key, and also contains both the plaintext and encrypted data managed by that password.

{length = 125} – inclusive length of entire outer block	2 bytes
{type = 1} – user access password protected data	2 bytes
{pt length = 45} – inclusive length of entire inner block	2 bytes
{aes-gcm iv} – initialization vector for auth/encrypt	12 bytes
{scrypt random salt} – update for password change	16 bytes
{scrypt log(n-factor)} – memory consumption factor	1 byte
{scrypt iteration count} – time consumption factor	4 bytes
{option flags} – 16 binary flags	2 bytes
{hint length} – number of chars in hint	1 byte
{pw verify sec} – seconds to run PW EnScrypt	1 byte
{idle timeout min} – idle minutes before wiping PW	2 bytes
{encrypted identity master key (IMK)}	32 bytes
{encrypted identity lock key (ILK)}	32 bytes
{verification tag} –	16 bytes

Figure 13: Type 1 Block Format

- Immediately following the type 1 block's two-byte length and two-byte type specifier is the two-byte "plaintext length" (pt length) of the unencrypted (plaintext) but authenticated data for this block. This is the length of the so-called "additional authenticated data" (aad) of the block's AES-GCM cipher. The length specification spans from the first byte of the block (the first byte of the whole block's length specification) up to, but not including, the beginning of the block's encrypted data (shown in **red** above).

- The 12-byte (96-bit) AES-GCM nonce appears next, immediately following the two-byte plaintext length. Note that '96 bits' is the natural length of aes-gcm IV nonces. These 96 bits are chosen randomly every time the block is re-encrypted since the encryption key will typically not be changing and it is crucial that IV nonces are never reused with the same key and differing data under AES-GCM. Since we might only be updating the plaintext user interface data with a short and fast password hint decryption, we would have no opportunity to establish a new key. So, a unique nonce will be used with the same quickly-decrypted key.

- Immediately following the AES-GCM IV nonce are the EnScrypt encryption parameters, the user's user-interface preferences and other non-critical information. Although it **CAN** be read without any authentication, all S[4] data is authenticated and it **MUST** be marked and regarded as untrusted until it has been authenticated under the user's local access password. In other words, prior to using any of the unencrypted data obtained here for any purpose important to security, the client **MUST** obtain the user's password and authenticate this data. Similarly, in order for any change made in this data to authenticate, the user's password will be required in order for an updated verification tag to be calculated. So, saving any changed settings may require the user's password or hint.

- Future versions of the SQRL S[4] specification may define additional plaintext to be appended to the end of the v1-specified plaintext, but individual clients may not arbitrarily store their own data there. That need may easily be fulfilled by creating, for example, a type 4 block for their own use.

- Encrypted data immediately follows the length-defined end of the inner plaintext block, extending to the end of the outer block (excluding the final 16-byte verification tag). The outer block's overall

length is specified at the top of the block. As before, future extensions of this specification may append additional encrypted data to the end of this list, but individual clients are not permitted to do so.

- *Operational Note:* This SQRL identity database bundles the GRC SQRL client user's user-interface preferences, including the time required to process a full password input. When this identity is imported into a compatible client running on a platform having a different processing speed - either a different PC or a mobile device - that SQRL client will obtain that user's preferred password processing time. Although different environments (for example, mobile) might suggest a differing security posture and password processing overhead, the client can note when a large difference exists between the processing time of the origin device - as revealed in the bundled time specification - and the current platform. The client can offer to immediately re-encrypt the user's just entered password to re-normalize all future password processing time.

Cross-Client Options & Flags

The type 1 secure storage block includes a set of operational options and settings which allow users to customize the SQRL client's operation to their preferences. These options are stored with the user's identity so that they are changed when the client's logon identity is changed, and so that they may follow the user from platform to platform and from device to device. If SQRL clients standardize upon these options, user convenience will be enhanced:

- hint length (characters)
  This one-byte value specifies the number of characters used in password hints. The default is 4 characters. A value of zero disables hinting and causes the SQRL client to prompt its user for their full access password whenever it's required.

- password verify (seconds)
  This one-byte value specifies the length of time SQRL's EnScrypt function will run in order to deeply hash the user's password to generate the Identity Master Key's (IMK) symmetric key. SQRL clients are suggested to default this value to five seconds with one second as a minimum. It should not be possible for the user to circumvent at least one second of iterative hashing on any platform.

- idle timeout (minutes)
  This two-byte value specifies the length of idle system time the hinting system is allowed to retain an abbreviated password before it is erased and the full password must be re-entered. This non-zero value, coupled with the 0x80 idle enable bit, are both required to enable idle timeout.

- option flags (default value: *0x01F3*)
  This two-byte value contains a set of individual single-bit flags corresponding to options offered by SQRL's user-interface. The individual bits have the following meaning:

  0x0001 Check for updates: This requests, and gives the SQRL client permission, to briefly check-in with its publisher to see whether any updates to this software have been made available. Note that GRC's client uses a low overhead DNS query to check for newer versions.

  0x0002 Update autonomously: This requests, and gives the SQRL client permission, to automatically replace itself with the latest version when it discovers that a newer version is available.

  0x0004 Request SQRL only login: This adds the "option=sqrlonly" string to every client transaction. The presence or lack of presence of this option string in any properly signed client transaction is used to push an update of a server-stored flag that, when set, will subsequently disable all traditional non-SQRL account logon authentication such as username and password.

0x0008 Request no SQRL bypass: This adds the "option=hardlock" string to every client transaction. The presence or lack of presence of this option string in any properly signed client transaction is used to push an update of a server-stored flag that, when set, will subsequently disable all "out of band" (non-SQRL) account identity recovery options such as "what was your favorite pet's name."

0x0010 Warn of possible MITM attack: When set, this bit instructs the SQRL client to notify its user when the web server indicates that an IP address mismatch exists between the entity that requested the initial logon web page containing the SQRL link URL (and probably encoded into the SQRL link URL's "nut") and the IP address from which the SQRL client's query was received for this reply.

0x0020 Discard password hint data upon blanking, suspend, etc.: When set, this bit instructs the SQRL client to wash any existing local password and hint data from RAM upon notification that the system is going to sleep in any way such that it cannot be used. This would include sleeping, hibernating, screen blanking, etc.

0x0040 Discard password hint data upon user switching: When set, this bit instructs the SQRL client to wash any existing local password and hint data from RAM upon notification that the current user is being switched.

0x0080 Discard password hint data after system idle: When set, this bit instructs the SQRL client to wash any existing local password and hint data from RAM when the system has been user-idle (no mouse or keyboard activity) for the number of minutes specified by the two-byte idle timeout.

0x0100 Warn of non-CPS authentication: When set, this bit instructs the SQRL client to notify its user whenever a non-CPS authentication is attempted. Since CPS provides extremely strong protection against website spoofing—which is a particularly significant concern for SQRL due to its high degree of automation and presumed automatic provision of security—this flag *must* default to enabled.

Type 2 - Rescue code encrypted data

The type 2 S^4 block supplies the EnScrypt parameters to convert a user-supplied "emergency rescue code" into a 256-bit symmetric key for use in decrypting the block's embedded encrypted emergency rescue code.

{length = 73}	2 bytes
{type = 2} – emergency rescue code data	2 bytes
{scrypt random salt}	16 bytes
{scrypt log(n-factor)}	1 byte
{scrypt iteration count}	4 bytes
{encrypted identity unlock key}	32 bytes
{verification tag} –	16 bytes

Figure 14: Type 2 Block Format

The type 2 block's unencrypted data, from and including the first byte of the block length up to but not including the first byte of the encrypted identity unlock key, is the unencrypted (plaintext) authenticated data for this block. Therefore, the length of the "additional authenticated data" (aad) of the block's AES-GCM cipher is 25 bytes. As with all of the S^4's blocks, the AES-GCM initialization vector (IV) is null (all zeroes).

Type 3 - Encrypted previous identity unlock keys

Since the type 3 S^4 block contains from one to four of the most recently replaced identity unlock keys (PIUKs), the presence of this block type in any SQRL identity is optional. It will only be present when a

identity has been rekeyed at least once. When present, this block is encrypted under the current identity's Identity Master Key (IMK). The IMK may be obtained either by decrypting the type 1 block with the user's identity passphrase, or by decrypting the type 2 block with the user's identity RescueCode then "EnHashing" the obtained Identity Unlock Key (IUK) to derive the Identity Master Key (IMK).

SQRL is designed to use extremely long-lived, hopefully perpetual, identity keying where the need to rekey any identity is vanishingly rare. Add to this the fact that authenticating to any site that recognizes a user through one of their previous identity keys autonomously updates that site to the users current key. This will tend to automatically keep websites' identity information current. These facts argue for a modest number of previous keys being sufficient to make the entire rekeying process transparent to all reasonable SQRL users (and even any who are unreasonable).

As shown below, the overall size (length) of the type 3 block will be 54, 86, 118 or 150 bytes depending upon whether it contains one, two, three or four previous identity unlock keys.

{length = 54, 86, 118 or 150}	2 bytes
{type = 3} – previous identity unlock keys	2 bytes
{edition >= 1} – count of all previous keys	2 bytes
{encrypted previous IUK}	32 bytes
{encrypted next older IUK (if present)}	32 bytes
{encrypted next older IUK (if present)}	32 bytes
{encrypted oldest previous IUK (if present)}	32 bytes
{verification tag} –	16 bytes

Figure 15: Type 3 Block Format

- The type 3 block's unencrypted 6 bytes of header data is the unencrypted (plaintext) "additional authenticated data" (aad) for this block. Therefore, the length of the "additional authenticated data" (aad) of the block's AES-GCM cipher is 6 bytes. As with all of the S^4 blocks, the AES-GCM initialization vector (IV) is null (all zeroes).

- This block contains from one to four most recently retired (previous) identity unlock keys (PIUKs). The up to four key slots are treated as an MRU (most recently used) list with the most recently retired keys placed at the top of the list and earlier keys "pushed" down, with the oldest key lost as it is replaced in the last position.

- The "edition" field is a monotonically incrementing 2-byte counter, incremented once with every identity rekeying. Being in the additional authenticated data header, it is not encrypted. Although this provides an indication of the identity's rekeying history, this information will be of no use to an attacker, whereas it will be extremely useful for the identity's owner, since it will allow them to clearly distinguish among same-name identities of differing editions. SQRL's paper identity printouts will also be able to display this edition number. If the edition were part of the block's encrypted data, the identity's passphrase would be required to view it, which would be inconvenient for its intended purposes.

The S4 Secure Storage System blocks assembled into a full identity store

SQRL's Secure Storage System (S^4) Encrypted Key Sample

```
sqrldata - lowercase signature means binary follows                8 bytes

  {length = 125} - inclusive length of entire outer block          2 bytes
  {type = 1} - user access password protected data                 2 bytes
  {pt length = 45} - inclusive length of entire inner block        2 bytes
  {aes-gcm iv} - initialization vector for auth/encrypt           12 bytes
  {scrypt random salt} - update for password change               16 bytes
  {scrypt log(n-factor)} - memory consumption factor               1 byte
  {scrypt iteration count} - time consumption factor               4 bytes
  {option flags} - 16 binary flags                                 2 bytes
  {hint length} - number of chars in hint                          1 byte
  {pw verify sec} - seconds to run PW EnScrypt                     1 byte
  {idle timeout min} - idle minutes before wiping PW               2 bytes

 {encrypted identity master key (IMK)}                            32 bytes
 {encrypted identity lock key (ILK)}                              32 bytes
 {verification tag} -                                             16 bytes

 {length = 73}                                                     2 bytes
 {type = 2} - rescue code data                                     2 bytes
 {scrypt random salt}                                             16 bytes
 {scrypt log(n-factor)}                                            1 byte
 {scrypt iteration count}                                          4 bytes
 {encrypted identity unlock key (IUK)}                            32 bytes
 {verification tag} -                                             16 bytes

 {length = 54, 86, 118 or 150}                                     2 bytes
 {type = 3} - previous identity unlock keys                        2 bytes
 {edition >= 1} - count of all previous keys                       2 bytes
 {encrypted previous IUK}                                         32 bytes
 {encrypted next older IUK (if present)}                          32 bytes
 {encrypted next older IUK (if present)}                          32 bytes
 {encrypted oldest previous IUK (if present)}                     32 bytes
 {verification tag} -                                             16 bytes
```

Figure 16: Assembled S4 Storage with All Three Blocks

S^4 Database header

SQRL's eight-byte 'sqrldata' file signature indicates that S^4 format blocks follow and the alphabetic case of the signature indicates whether the balance of the file is straight binary or 6-bit base64url encoded. The eight-byte header stands alone and does not participate in the authenticated encryption of the blocks that follow.

```
sqrldata - lowercase signature means binary follows                8 bytes
```

Figure 17: S4 Storage Database Header

Uniqueness

Since no identity should logically contain more than one instance of each block type, no client should ever generate an identity having more than one Type 1, 2 or 3 block. Any client encountering an identity containing more than one of either type should reject the entire identity as suspicious and invalid. Clients wishing to tolerate aberrant identities may choose which one of the duplicated block types to honor.

Expounding on expansion

• Some observers have suggested that peripheral features of the underlying SQRL protocol, such as the use of GRC's proposed EnScrypt PBKDF should remain separate. This S^4 database format provides

for the addition of any other type of password authentication by leaving all other block types open and available. And the compact representation of this database leaves ample room for future parallel authentication or alternatives.

• If SQRL should evolve in the direction of a form-fill application, that private and probably encrypted data could occupy currently undefined block types without duplicating any of SQRL's existing data. A future encrypted and authenticated block could implicitly infer the use of, for example, the encryption parameters obtained from the existing type 1 block.

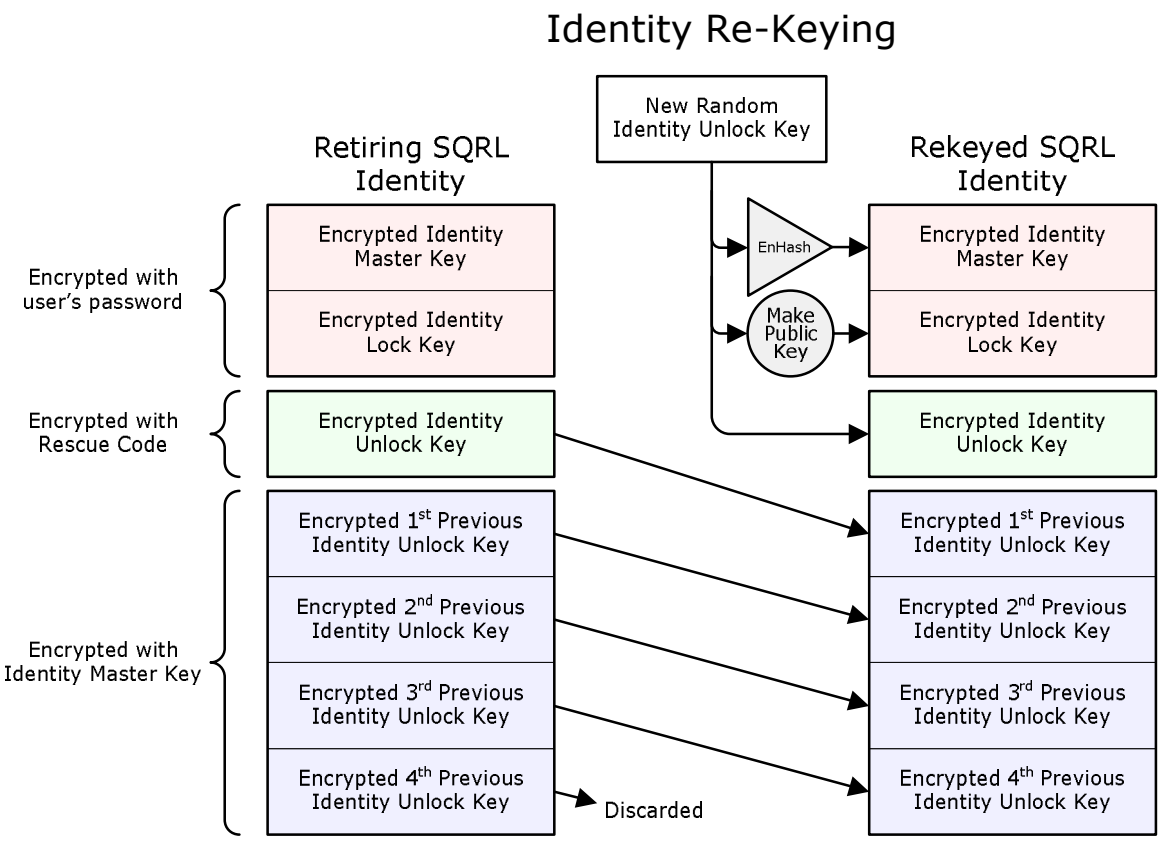

Figure 18: Identity Re-Keying Key Migration

Armed with an understanding of the layout, format and operation of SQRL's S^4 Secure Storage System the diagram above should be understandable. It depicts how SQRL handles autonomous identity re-keying.

Identity Re-Keying Point by Point

- SQRL users may re-key their identities at any time. The SQRL system is designed to protect its user's identity so that re-keying should only rarely be necessary. But a system that is unable to accommodate the possibility of re-keying would not be useful.
- As shown in the diagram above, the primary event of rekeying is that the current Identity Unlock Key (IUK) stored in the Type 2 identity block is moved to the top of the list of possibly previous IUKs stored in the Type 3 identity block, and any previous IUKs stored in the Type 3 block are pushed down the 4-deep list with a possibly 4th previous IUK pushed off the end and forgotten.
- The current IUK residing in the Type 2 block is encrypted under the current identity's Rescue Code, whereas any previous IUKs stored in the Type 3 block are encrypted under the current Identity Master Key (IMK). This re-encryption and relocation of the current IUK into the previous IUK list, encrypted under the new IMK makes the previous IUKs available to the client when its user only has their password.

- A completely new identity is then created, exactly mimicking the initial identity creation, which assigns new Identity Unlock, Identity Master and Identity Lock Keys as shown above.
- At this point, after the re-keying, the user has a completely new identity with the immediately previous IUK now moved to the top of the identity Type 3 block where successively older previous IUKs may also be present.

Authenticating with Prior Identities

Any SQRL client whose identity is carrying one or more previous identity keys could be known to a SQRL server by its current identity or by one of those previous identities. Such clients present **both** their current and their most recently re-keyed identities when authenticating to a remote SQRL server. "Presenting both identities" means sending each identity's public key and co-signing the client's query with each of the respective private keys.

The SQRL server's response will indicate whether it has recognized the user by their current or previous identity, or neither. If the response is "neither" and the SQRL client has an older, not-yet-offered previous identity, it will reissue its query using both its current and the next-less-recent previous identity. The SQRL client will continue trying successively older previous identities until the server either recognizes one of the identities or the client runs out of previous identities.

When a SQRL server recognizes a user by a previous identity, for any commands other than "query", it will always autonomously and immediately update its stored keying material to the current identity that the SQRL client also provided.

Server-Side Crypto

One of the significant benefits of the SQRL system is that nearly all of the heavy lifting work is performed by SQRL clients. This is significant for SQRL's deployment because it means that implementing server-side support is quite simple. SQRL servers have only three simple cryptographically related tasks:

Generate Secure SQRL Nuts

SQRL nuts serve as nonces to force the signatures of unique envelopes and to prevent brute-force nut guessing attacks. They also prevent replay attacks because each issued nut can only be used once. The SQRL server therefore must produce a series of nuts in a sequence which cannot be guessed or predicted from external observation, and each nut issued must be invalidated after its use to prevent nut reuse.

Our favored solution for nut sequence synthesis is to symmetrically encrypt a 64-bit monotonically increasing counter with an internally secret key:

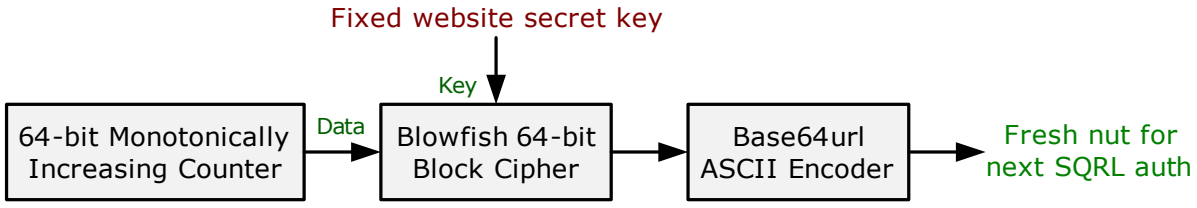

Figure 19: Algorithmic Nut Synthesis

A 64-bit nut-space provides ample protection from brute-force guessing while keeping nuts short after they have been encoded for use into textual ASCII form with a base64url encoder.

However, this solution is not suited to large distributed server farms where the maintenance of a single counter would be inconvenient. For distributed nut generation, using 64 independent bits of

high-quality entropy is probably sufficient. The chance of 64-bit nut collision is very small, and since nuts are relatively short lived and are also monitored for single use, 64-bits would be sufficient.

It is **crucial** that issued nuts are placed onto some form of valid-nut list and are removed when they are returned by any client, whether or not the client's query succeeds. Nut reuse must always fail.

Authenticate Client-Returned Data

As was also noted earlier, SQRL servers must guard against any modification of the query response they sent to the client. Since all SQRL clients return the server's previous query response with any next query, SQRL servers **must** verify that what they have received back, signed by the client, is exactly that they previously returned to the client. This can be accomplished in a minimally stateful way, at the expense of increasing the size of their response, by appending a standard message authentication code (MAC) to the reply and verifying it in the return. The HMAC-SHA-256 function could be used from the Sodium library with an internal server secret to generate the MAC.

Alternatively, SQRL servers might take a simple SHA-256 hash of their client response and store it in a "pending authentication" structure for subsequent verification upon return.

Verify Client Signatures

The final server-side responsibility is the verification of the signature of every SQRL client query. Every SQRL client query provides both one or more client public keys and one or more privately signed signatures which are verifiable with its matching public key. Every signature provided by the SQRL client for which there is a provided public key **must** be verified by the server, and any failure in any signature **must** abort the client's query and produce no change of the client's account state on the server.

The Sodium library provides a convenient elliptic curve signature verification function for this purpose:

```
int crypto_sign_open(unsigned char *m, unsigned long long *mlen_p,
                     const unsigned char *sm, unsigned long long smlen,
                     const unsigned char *pk);
```