Security Proof for the Tabby PAKE Protocol

Password Hashing Competition Submission Document

Designer and Submitter

Christopher A. Taylor, MSEE
mrcatid@gmail.com

March 30, 2014

Latest information available online:

http://www.tabbypake.com

Abstract

The Password Hashing Competition is concerned with improving the handling of sensitive password information. While Tabby does not provide a new approach for password hashing directly, it does improve on the state of the art for mutual authentication based on passwords. Therefore, it was suggested by an anonymous referee that Tabby should join the conversation on new directions in password-based security.

This document motivates, concretely describes, and validates the security claims of the Tabby PAKE protocol, which provides mutual authentication via e.g. shared secret PIN or password typically in a network setting. Tabby PAKE is an augmented Password Authenticated Key Exchange protocol in two rounds with forward secrecy. It is defined for conservative and fast elliptic curve math at a 125-bit security level, well-matched to modern encryption techniques.

It is based on the patent-free SPAKE2 protocol by Abdullah, which was improved by Boneh into an augmented scheme called PAKE2+. Tabby PAKE further improves PAKE2+ to fix the backdoor issue by applying the patent-free Elligator algorithm introduced by Bernstein et al. Tabby will remain patent-free and royalty free.

A security proof is provided in this document to motivate its use in practical applications, and a reusable software implementation is available online now.

Tabby PAKE server operations take 100x less time than SRP server operations, while providing a much higher security level and running in constant-time to avoid timing side-channel attacks. The password hash itself is allowed to take arbitrary amounts of time on the client side and does not impact server performance. As a result, Tabby PAKE is fast enough to be practical, and will hopefully improve the security of many applications that need password-based authentication.

1 Table of Contents

2 Introduction

2.1 Motivation

Authentication should be mutual in network communications. When a thin network client authenticates with a remote server, both sides of the authentication step should come out of the exchange with enough proof to trust the other side.

Traditional access credentials for single users are often categorized:

- **Something you have**: Smart Card containing a secret key, Private key for a corresponding public key and a certificate for that public key
- **Something you know**: User name, User password, PIN
- **Something you are**: Biometrics (probably out of scope)

Applicable to mutual authentication, access credentials for the server machines can also be usefully categorized:

- **Something you have**: Private key for a corresponding public key and a certificate for that public key
- **Something you know**: Direct read access to user account database

The distinction between "something you have" and "something you know" is in the amount of entropy. Private keys (something you have) typically have about 256 bits of entropy, enough to directly deter brute-force. English phrases typically have about 4 bits of entropy per character, and passwords have about 6 bits of entropy per character when uniformly randomly chosen. The overall entropy of passwords (something you know) is often fairly low.

Longer passwords may be considered weaker precisely because they are more likely to be written down and become "something you have" instead.

In most systems, all of the server credentials are provided by a fragile public key infrastructure. The certificate chains that currently establish trust in public keys are under constant attack and appear to be untrustworthy, prompting a move towards peer-to-peer Certificate Authorities. As another example, Secure Shell (SSH) clients typically assume that the server's public key has not been tampered with. This is problematic because on the first connection to the server, the public key is not signed and no trust can be established.

In most systems, all of the client credentials are provided by either a password or a certificate, but usually not both. This is also problematic as either of these credentials may be stolen.

If a password hash is revealed (e.g. from a compromised database), then password strengthening may be used to make brute-force more difficult. The algorithms for password strengthening will undoubtedly be improved as a result of the Password Hashing Competition. However hashes of PINs are trivial to brute force regardless of strengthening. In both cases, an active or passive attacker should not be given access directly to the password hash unless the account database is compromised. Instead, a password authentication protocol should be employed so that attackers are unable to directly attack the hash via the login process.

Furthermore, the server itself should not be trusted unless it can demonstrate knowledge of the password, and the server should also not be given access to the password hash by a client that is logging in. The desire for two-factor mutual authentication with untrusted servers motivates password authentication provided by a zero-knowledge proof (ZKP). With a ZKP, both sides can demonstrate knowledge of the password without providing any way to brute-force it aside from repeated connection attempts. Moreover access to the account database will not allow impersonation without brute-forcing the password, which is referred to as the "augmented" property. The mutual trust established by password authentication can cement trust in the server's public key certificates for future communication.

2.2 Background

Over the past 18 years there has been an explosion of solid password authentication protocols proposed that have security proofs and good analysis. For brevity just a few of them are presented here:

B-SPEKE[1] was one of the first augmented schemes, developed by Jablon in 1997. It is an augmented protocol using large finite field math that takes three protocol rounds to achieve mutual authentication. SPEKE is patented until 2017[2] by Phoenix Technologies.

SRP[3] was developed by Wu in 1997 at Stanford. SRP circulated on *sci.crypt* until it was called SRP-3, and was

refined again into the latest incarnation SRP-6. SRP-6a has been found in unusual places such as the World of Warcraft login protocol. It is an augmented protocol using large finite field math that takes two protocol rounds to achieve mutual authentication. SRP is patented by Stanford, though it is free to use. SRP is considered the standard for password authentication security, and is likely still the best option to use at this time.

SPAKE2[4] was first proposed by Abdalla and Poincheval in 2005. Until now it is the only high-speed elliptic curve PAKE of which this author is aware. It has been found in the Chromium source code. PAKE2+ by Dan Boneh is a simple modification of SPAKE2 that has the augmented property. These are two-round protocols that achieve mutual authentication using elliptic curves. SPAKE2 and PAKE2+ are both patent-free. SPAKE2 is slightly more efficient than Tabby but it is not augmented, and PAKE2+ is slightly less efficient than Tabby.

The major disadvantage of SPAKE2 and PAKE2+ is that these schemes have a built-in backdoor: The security depends on a constant elliptic curve point that is generated by an unknown scalar. This point is agreed upon in advance by all users of the protocol with the assumption that no one knows its private key. As another result, the security of every protocol instance depends on a single ECDLP problem being unsolvable.

AugPAKE[5] was first publicly disclosed by Shin in 2008. AugPAKE has been developed into a TLS proposal[6]. It is an augmented protocol using finite field math that takes two protocol rounds to achieve mutual authentication. AugPAKE is patented[7] by AIST, though it is free to use. In practice, AugPAKE is slightly faster than SRP and has a simpler proof of security.

Despite the assurances made by the AugPAKE authors it is not possible to implement AugPAKE using elliptic curve math. The group operations in elliptic curve math are the following, using their TLS proposal notation where q is a prime curve order, and multiplication is point scalar multiplication:

$$X = G \cdot x$$
$$Y = G \cdot x \cdot y + G \cdot w \cdot r \cdot y$$
$$z = \frac{1}{x + w \cdot r} \bmod q$$
$$K = Y \cdot z$$

While it may work in the finite field case, this expression for K clearly does not simplify down to the desired $K = G \cdot y$ in the elliptic curve setting. This may explain why no elliptic curve implementations of AugPAKE exist.

J-PAKE[8] was first proposed by Hao and Ryan in 2010. J-PAKE is proudly not an augmented scheme, with the authors citing complexity. It uses elliptic curve math in a protocol that takes three protocol rounds to achieve mutual authentication. It is explicitly patent-free. Performance of J-PAKE was not measured while benchmarking Tabby, but should be ~10x slower due to the many additional curve operations required . The J-PAKE paper claims SPEKE execution time is similar.

Note that there are only two elliptic curve algorithms in this list. The reason for this is that elliptic curves do not admit a simple construction useful for PAKE. An obscure primitive operation called a “full domain hash function” is required to achieve the same simplicity that SPEKE did in 1996. A full domain hash function that runs efficiently and in constant-time was recently introduced by Bernstein, Hamburg, Krasnova, and Lange in 2013 as part of their Elligator[9] system. Tabby adapts the Elligator full domain hash function for PAKE as suggested by Mike Hamburg[10] on the CFRG mailing list.

2.3 Tabby PAKE Security Claims

1. **Off-line dictionary attack resistance**: Passive and active attackers must not gain any knowledge that enables them to mount an exhaustive search for the password by interacting with the protocol.
2. **Forward secrecy**: The produced shared secret key for prior instances of the protocol should remain confidential even if the password is later compromised.
3. **Known-session security**: If a shared secret for one session is compromised, it does not also lead to a compromise of the password, nor of any other sessions.
4. **On-line dictionary attack resistance**: Only one password may be tested per execution of the protocol.
5. **Timing side-channel attack resilience**: All algorithms run in constant-time with respect to secret information in the protocol.
6. **Augmented**: The verifier stored in the account database cannot be used to impersonate any user unless the original password is found by exhaustive search.
7. **Low computation cost**: The real execution time required on the server should be similar to the cost of elliptic curve Diffie-Hellman key agreement in order to not enable denial-of-service attacks.
8. **Minimal rounds**: Requires the minimal number of rounds (2) to reach mutual authentication.
9. **Patent-free**: Avoids existing patents and promises explicitly to never impose a patent on the design.
10. **Security proof**: A security proof is provided as an argument that it is a reliable mechanism to use for real applications.
11. **125-bit security level**: Provides security equivalent to exhaustive search of a 125-bit key, for future proofing.
12. **Backdoor-free**: There is no place to hide a backdoor in the specification.

It is notable that the Tabby PAKE is the only algorithm that achieves all of these goals and sets new speed records for a practical implementation that can be downloaded and incorporated into existing software today.

The author, Chris Taylor, does not claim that any of the ideas presented here are original. Every component of Tabby is based on existing work by far more diligent and far more brilliant people. The primary intent of presenting the reader with this construction is to enable building useful software that has a chance of improving the state of password security by leveraging the hard and generous work of many others.

3 Tabby Protocol

3.1 Definitions

3.1.1 Symbol definitions

Mathematical operators are defined for elliptic curves:

- P + Q refers to the summation of two points on the elliptic curve, yielding a third point.
- P - Q refers to the subtraction of two points on the elliptic curve, yielding a third point.
- xG refers to the multiplication of the generator point G on the Snowshoe curve by the scalar x in the range [1, q-1].
- xP refers to the multiplication of the point P on the Snowshoe curve by the scalar x in the range [1, q-1].
- Point variables are named with upper-case letters.

Mathematical operators are defined for scalar values:

- x + y refers to the summation of x,y modulo the Snowshoe curve order q, a large prime.
- $x \cdot y$ refers to the product of x,y modulo the Snowshoe curve order q.
- Scalar variables are named with lower-case letters.

Other operators relate to data in bytes:

- x || y refers to the concatenation of x and y, where x is explicitly placed in the low bytes and y is placed in the higher-address bytes.
- f(x, y, z) refers to a function taking on several parameters x, y, z. These functions are defined below and can have differing numbers of parameters and names in place of "f."

Please see the Snowshoe section of the document for full information about the elliptic curve group.

3.1.2 PBKDF

PBKDF is defined as a password-based key derivation function given a password and a salt, which produces a 64 byte output.

For this initial Tabby PAKE specification, an instance of the PBKDF function is chosen that is somewhat precursory since the PHC has not come to its conclusion yet. At the end of the competition, the best suitable PBKDF will be chosen for use in Tabby PAKE.

PBKDF is Lyra[11] with parameters:

- saltSize = 16 bytes
- timeCost = 2 (# of iterations)
- blocksPerRow = 64 (# of 64 byte hash blocks per matrix row)
- nRows = 3000 (# of 4 KB rows => 12 MB)
- Output = 64 bytes

This runs in about ~100 milliseconds on a laptop. The selection of T(timeCost) is based on recommendations by Solar Designer[12], and the rest are due to private communications.

3.1.3 H

H is defined as a secure hash function that takes variable-length input and produces 64 bytes of output. The function must be a secure PRF. For this initial Tabby PAKE specification, an instance of the H function is chosen to be the BLAKE2b hash function, which does aim to be a PRF.

BLAKE2b runs exceptionally fast on 64-bit Intel servers, which is where performance needs to be best: The client operations are slow by design, but the server operations have the opportunity to be fast.

3.1.4 GeneratePasswordVerifier

The GeneratePasswordVerifier function takes as input:

- Salt (16 bytes)
- Username (variable length) is case-sensitive
- Realm (variable length), used to partition passwords to be service-specific
- Password (variable length) is also case-sensitive

And produces as output:

- Client secret (32 bytes)
- Password verifier (64 bytes)

The GeneratePasswordVerifier algorithm:

Step 1. Validate input parameters, emitting a failure code on error.

The username length and password length must be at least one character.

Step 2. pw = H(username || " : " || realm " : " || password)

The value pw is a 64-byte buffer on the stack.

Step 3. Erase internal hash state from the stack.

Step 4. v = PBKDF(salt, pw)

The value v is 64-byte buffer that is the Client secret.

Step 5. Treat v as a 512-bit random integer and reduce it modulo q.

This is implemented by the Snowshoe library in constant-time. The reasoning for using an input of 2^{512} states to produce a key with only about 2^{252} states is to avoid biases in the key while making full use of the key space.

Step 6. V = vG

Step 7. Return V in affine coordinates as the password verifier, and v as the client secret.

The Snowshoe library is used for Generator-base point multiplication, and the result V is the password verifier. This operation can fail if the input value v is 0. This is extremely rare, and it causes the function to fail just like an invalid input. The user of this function will know to expect this rare error case and select a new salt value. While this would cause a timing attack vector if it was anywhere near a frequent occurrence, this probably will never happen: The odds are roughly 1 in 2^{252} executions of the function.

The client secret is used by the client side to produce its zero-knowledge proof of the password. The password verifier is used by the server side. The asymmetry of the elliptic curve operation prevents the password verifier from being used to log in.

This operation is dominated by the expensive PBKDF function, so the other operations do not need to be fast, but they do need to run in constant-time.

Using keys uniformly distributed in [1, q-1] is important for Tabby's security proof. Some other implementations of ECC have a faster key generation process where they will just mask off some bits (e.g. Ed25519) on a random value, which gives away a few bits of security in return for faster and deterministic key generation. Tabby's approach is also practically deterministic (aside from the very remote chance of zero

popping up) and offers a higher security level, which makes up for the "security level bits" lost by using an efficient endomorphism to speed up curve math. The extra modulo operation is not much slower in practice (<2% overhead) than masking. And key generation is rarely a bottleneck.

3.1.5 ElligatorDecode

ElligatorDecode refers to the Snowshoe Elligator point decoding algorithm. It takes as input a 255-bit value as input and returns a point on the curve. The range of the function is all points on the curve.

The domain of the Elligator point decoding function from the original article by Bernstein et al contains values in the interval $[0,\frac{(2^{127}-1)^2+1}{2}]$ rather than all points on the curve. The Snowshoe library improves this situation by flipping the sign of the X coordinate based on the input, which is described in full in the Snowshoe details later in this document. This small improvement turns the Elligator point decoding function into a true full domain hash function.

There is still some potential bias in terms of which points are selected because the number of points on the curve $\#E = 4q = 2^{254} - d$, where d =

47773122019030021920049591544028017290 8 (a 129 bit number)

is the deviation from a power of two. For a 255-bit uniform binary string input there is bias proportional to the ratio $\frac{d}{2^{255}} \approx \frac{2^{129}}{2^{255}} = \frac{1}{2^{126}}$, small enough to be ignored in practice. This is the same logic used to validate the claimed lack of bias in the EdDSA spec.

3.1.6 ElligatorEncrypt

ElligatorEncrypt refers to an algorithm based on Snowshoe operations:

The ElligatorEncrypt algorithm

Given a uniformly random scalar b in the range [1, q-1] and a point E presumed to be selected by the ElligatorDecode function, produce point B' that is an encrypted representation of point B=bG.

Step 1. B = bG

Multiply the scalar by the generator point on the Snowshoe curve to produce point B of order q. The point B is not reduced to affine coordinates yet for efficiency.

Step 2. B' = B + E

The projective form of the points B, E are preferred in this step to avoid multiple reductions to affine form.

Step 3. Return B' in affine form.

3.1.7 ElligatorSecret

ElligatorSecret refers to an algorithm based on a few simple Snowshoe operations. It takes as input:

- a: Scalar in the range $[1, q-1]$
- B': Any point on the curve where $x \neq 0$
- E: A point presumed to be selected by the ElligatorDecode function
- b: [Optional] Scalar in the range $[1, q-1]$

- V: [Optional] Any point on the curve where $x \neq 0$

As output it produces a 64-byte secret point on the curve.

The ElligatorSecret algorithm

Step 1. $B'' = B' - E$

This should recover the original value $B = B''$ if the input B' is valid. Step 1 is effectively "decrypting" the original point B similar to the EKE scheme.

Step 2a. $R = a \cdot B$

Step 2b. $R = a \cdot B + b \cdot V$ [optional]

Step 2 can optionally simultaneously multiply two scalars by variable base points. This optional form is used on the server to speed up the computations required for the augmented property. Note that SPAKE2 would only require a single point multiplication, but PAKE2+ adds a second point multiplication. Tabby PAKE combines the two point multiplications of PAKE2+ for better performance.

Step 3. Return R in affine form.

3.2 Protocol Specification

3.2.1 Account Creation

In order to process logins, the server will need a (salt [16 bytes], verifier [64 bytes]) pair to associate with the username of the account. During account creation, a random salt is generated, and the GeneratePasswordVerifier() function is run to generate the password verifier. The resulting (salt, verifier) pair can be stored for authentication.

This account creation step can run on the client or on a dedicated server providing a web API. Ideally this will not run on a server, so that the server side has no access to the original password. When running on the client, a trusted server public key can be used to establish a secure tunnel for transmitting the password verifier.

3.2.2 Changing Passwords

After authentication completes, the server can accept a password change request from the client. The client can select a random salt and then execute the GeneratePasswordVerifier() function to produce a verifier and transmit the (salt, verifier) pair to the server for storage. This relies on the strength of the account authentication step to provide security against account compromise through the password changing mechanism.

3.2.3 Tabby PAKE Protocol

Two parties, Client and Server, wish to mutually authenticate over an insecure communication channel using a pre-established password verifier available to the server.

Client is given: [username(string)], [password(string)], [realm(string)].

Server has access to a database mapping usernames to password verifiers and salts.

Both sides also have a string called ExtraTags that may contain a public key for each side or any other identifying information.

Round 1

Client transmits:

- [**Username** (string)]

Server Online Processing:

Step 1. Query database for username: [verifier V(64 bytes)], [salt(16 bytes)]

Step 2. T = Low 32 bytes of H(V || salt)

Step 3. E = ElligatorDecode(T)

Step 4. Choose random x in [1, q-1].

Step 5. X' = ElligatorEncrypt(x, E):

X = xG

X' = X + E

Step 6. Store X', E, x, V for later.

Server transmits:

- [**Salt** (16 bytes)]
- [**X'** (64 bytes)]

Client Online Processing:

Step 0. Client validates the point X' to ensure that it is on the curve and that the X coordinate $\neq 0$. The protocol is aborted on any failure.

Step 1. (v, V) = GeneratePasswordVerifier(username, realm, password, salt)

Step 2. T = Low 32 bytes of H(V || salt)

Step 3. E = ElligatorDecode(T)

Step 4. Choose random y in [1, q-1].

Step 5. Y' = ElligatorEncrypt(y, E):

Y = yG

Y' = Y + E

Step 6. h = H(X' || Y')

Step 7. $a = (y + v \cdot h) (\mathrm{mod}\, q)$

Step 8. Z = ElligatorSecret(a, X', E):

X = X' - E

Z = aX

Step 9. [K(64 bytes)] = H(E || X' || Y' || Z || ExtraTags)

Step 10. [CPROOF (32 bytes)] || [SPROOF' (32 bytes)] = H(K)

Step 11. Store SPROOF' for later.

Round 2

Client transmits:

- [**Y'** (64 bytes)]
- [**CPROOF** (32 bytes)]

Server Online Processing:

Step 0. Server validates the point Y' to ensure that it is on the curve and that the X coordinate $\neq 0$. The protocol is aborted on any failure.

Step 1. h = H(X' || Y')

Step 2. $b = (x \cdot h) (\mathrm{mod}\, q)$

Step 3. Z = ElligatorSecret(x, Y', b, V, E):

Y = Y' - E

$Z = x \cdot Y + b \cdot V$

Step 4. [K(64 bytes)] = H(E || X' || Y' || Z || ExtraTags)

Step 5. [CPROOF' (32 bytes)] || [SPROOF (32 bytes)] = H(K)

Step 6. Verify that CPROOF' matches the received CPROOF in constant time. Abort on mismatch.

Step 7. Erase all stored variables aside from K.

Server transmits:

[**SPROOF** (32 bytes)]

Client Online Processing:

Step 1. Verify that SPROOF' = SPROOF in constant time. Abort on mismatch.

Step 2. Erase all stored variables aside from K.

Mutual authentication has been achieved at this point, and both sides share a 64-byte secret key K.

3.3 Discussion

3.3.1 Tabby PAKE within a secure tunnel

Tabby PAKE can be used for password-based login within a secure tunnel. Since the username is sent in the clear, it is preferred that a secure tunnel is established prior to the Tabby PAKE protocol so that in some cases the combination can provide deniability as well. Because the client goes first, deniability cannot be guaranteed if PKI cannot be used to protect the username. It seems impossible to provide better username deniability without adding more protocol rounds.

When used within a secure tunnel, the public key of each side of communication must be passed in as the ExtraTags. If the ExtraTags do not include the server's public key and the secret key K is not used to secure further communication (ie. only for authentication), then a Man-in-the-Middle attack is possible: The attacker would be able to proxy the Tabby PAKE protocol through a second tunnel made with the real server. Both sides would reach mutual authentication. By mixing in the actual server's public key on the client side, this attack is defeated. It is recommended to use the shared secret key K to key an AEAD scheme to exchange secure messages after login so that the protocol is harder to misuse in this way.

3.3.2 Importance of the augmented property

The augmented property guarantees that the server with access to V cannot also act as a client with access to the plaintext password without first launching an exhaustive search for the password.

While this is a special property for PAKE schemes, it would be considered a security regression if it was not provided. For instance, if a server has access to a database of password hashes, then the client can provide the plaintext password and the server can verify the password hash matches the stored hash. But an attacker with only access to the password database cannot log in without first cracking the password hash. Without the augmented property, PAKE schemes lose this security guarantee, and the password verifier could be used to log in without the need for password cracking. Therefore the augmented property is essential.

3.3.3 Tabby PAKE key agreement

Tabby PAKE can alternatively be used for key agreement as a secret key K is provided to each party. This key has the forward secrecy property, meaning that if the user's password is later compromised the secret key will remain secret. However there is no guarantee of deniability, as the username is sent in the clear and a passive observer can tell if the user was able to be authenticated.

3.3.4 Extra resistance to cracking

When the password database is compromised in some way, dictionary attacks can be used to recover the passwords.

For servers running the Tabby PAKE protocol, the password verifier V is stored, which is related to the password via V = vG, where v = PBKDF(salt, pw) mod q.

Conducting an exhaustive dictionary-based search to recover the password is the best attack given this data. This is essentially password cracking with the added step of performing elliptic curve point multiplication. This extra step may help strengthen the password hashes against cracking in similar ways to a PBKDF because the elliptic curve math is fairly complex and would make the task of developing an efficient password cracking tool much more difficult.

3.3.5 Discussing protocol tweaks

It is important that the client and server do not accept any messages aside from the password login messages until mutual authentication is reached. However, some changes can be made to the protocol that may be interesting.

While this is a two-round protocol, it is clear "false start" is possible: The client can start sending authenticated data along with its CPROOF, effectively making it a one-round protocol in some situations.

If the salt is removed from the protocol, then the client is able to send data earlier, which would remove one

message from the protocol, and the server would send a proof before the client. There are pros and cons to having the server prove knowledge first. In both cases, both sides get to guess just one password. However, all sources of information on password security indicate a salt twice as large as the birthday attack bound is essential, so it should not be removed.

3.3.6 Possibilities for side-channel exfiltration

The Tabby PAKE protocol may be abused in order to exfiltrate sensitive data from a compromised system.

To briefly summarize the transmitted data:

- C2S [Username (string)]
- S2C [Salt (16 bytes)] [Server ephemeral point (64 bytes)]
- C2S [Client ephemeral point (64 bytes)] [CPROOF (32 bytes)]
- S2C [SPROOF (32 bytes)]

If Server and Client are not actually running the protocol, a third party is unable to verify that any of the data is valid except that the points are actually on the Snowshoe curve. So it is trivial to hide information in any of these fields. However it would be much easier to legitimately run the protocol if it is allowed, and then exchange any exfiltrated data in encrypted form. It may also raise red flags if the protocol is being run many times as it is intended to only be run once. And also it is best to run the protocol within another layer of encryption to attempt to provide username deniability.

If the Server and Client are legitimately running the protocol and are not just faking the exchange, then it becomes much harder to hide any information for exfiltration. The username must match a user in the database, which can conceivably be used to encode some bits of information if the username is selected from a dictionary.

The salt is fixed for each user and so does not offer any additional information for use in exfiltration.

The server ephemeral point could be chosen to encode some information. For example, random ephemeral points can be generated until the low bits of the X coordinate take on some desired value. And the same is true of the client ephemeral point.

The client proof and server proof may similarly be used to transmit information by brute-forcing the client ephemeral point to yield a proof value that has selected low bits, though it appears that it is easier to hide information in the point itself.

3.3.7 Discussing some near-attacks

There is a potential flaw in this protocol that leads to an offline dictionary attack from the server's first response of X': X is always of order q, but Elligator generates points E of order 4q. And the sum is sent in the clear. So for example, if X' = X + E is of order q, then an attacker can eliminate all passwords that do not lead to a point of order q. The approach taken to fix this is to multiply the Elligator output by 4, which guarantees that the result is a point of order q.

The introduction of ***h*** is important to maintain the augmented property against specially crafted values of ***Y'***. In an earlier version of this protocol, the server calculated $Z = x(Y' - E + V)$ and the client with knowledge of ***V*** could send $Y' = Y + E - V$, which would make the server evaluate $Z = x(Y + E - V - E + V) = xY$, so the attacking client would just need to calculate $Z = yX$ and could impersonate any user. PAKE2+ by Dan Boneh fixed this issue by computing $Z_1 = xY$ and $Z_2 = xV$ separately.

To fix the issue more efficiently, the server in Tabby PAKE computes $Z = x \cdot Y + x \cdot h \cdot V$, with $h = H(X' \| Y')$. This simultaneous evaluation takes about 25% less time than performing two point multiplications.

To attempt to attack this new, fast construction, the value for ***h*** can be selected through brute force by the client via the choice of ***Y'*** to choose ***h*** such that $Y' = Y + E - h \cdot V$, but this would have a higher difficulty than just brute-forcing a 256-bit hash and would need to be done in real-time since ***h*** incorporates the per-session random value ***X'*** from the server. As a result the attack is impractical. It is nice that no additional randomness needs to

be introduced to solve this problem.

On the subject of handling random inputs, all random inputs are used as secret scalars for elliptic curve point multiplication. And where the secret scalars are added together directly they are mixed with a hash that is based on these public points. This is the strongest way to use random inputs because an attacker able to backdoor the random number generator inputs must control **all** the bits, which is likely impossible to do with the Cymric generator. As a result, Tabby PAKE is robust against even some sophisticated attacks targeting its random number generator. This type of attack is discussed further in Bernstein's "Entropy Attacks!"[13] blog.

3.4 Security Proof

3.4.1 Dictionary attack resistance

Off-line dictionary attacks involve exhaustive search for the password based on any revealed information. The main purpose of Tabby PAKE is to prevent an active or passive attacker from being able to search for the password in this way.

On-line dictionary attacks can be launched during the protocol, and include all classes of attack where a password can be crossed off a list of possible passwords. Tabby PAKE guarantees that only one password can be eliminated by a malicious client or server for each protocol instance, forcing an incredibly slow exhaustive on-line search for the password that can be practically rate-limited.

Rather than retreading over the same thorough work of Abdalla and Pointcheval, it is referenced. The security proof for SPAKE2[14] is directly referenced here. SPAKE2 is secure in the random oracle model and no attempts to improve the proof are made here.

The approach taken is to introduce each modification to SPAKE2 and indicate why the original security proof is not invalidated.

First modification:

N^pw, M^pw is replaced with a single base point E^pw.

In SPAKE1,

$$K_A \leftarrow (Y^* / N^{pw})^x = (Y \cdot N^{pw} / N^{pw})^x$$
$$K_B \leftarrow (X^* / M^{pw})^y = (X \cdot M^{pw} / M^{pw})^y$$

becomes

$$K_A \leftarrow (Y \cdot E^{pw} / E^{pw})^x = G^{xy}$$
$$K_B \leftarrow (X \cdot E^{pw} / E^{pw})^y = G^{xy}$$

The security proof still works, although the probability of collision between transcripts increases accordingly, while still being negligible.

Second modification:

E^pw is defined as ElligatorDecode(pw).

Elligator is taken to be a full-domain hash function, meaning that in the random oracle model it maps each password to a unique point of order q, for which the private generator scalar is unknown. All properties of the original construction E^pw used by the SPAKE2 proof are maintained.

Third modification:

To add the augmented property, PAKE2+ adds a second key that gets mixed into the hash:

$$K_{A2} \leftarrow V^x = G^{xv}$$
$$K_{B2} \leftarrow X^v = G^{xv}$$

In the random oracle model, the ideal PRF hash will properly mix in this additional key material and only strengthen the security proof of SPAKE2.

At the same time, the CDH assumption protects the value v from compromise.

Fourth modification:

Instead of two separate key pairs (K_A, K_{A2}) and (K_B, K_{B2}), a combined key $Z_A = K_A + h \cdot K_{A2}$ and $Z_B = K_B + h \cdot K_{B2}$ is calculated by each side for efficiency reasons. The value h is selected as PRF(X^* || Y^*), and is proven below to effectively randomly mix the two keys. The resulting bound on the proof is raised slightly by the tiny $\frac{1}{q-1}$ probability of a collision.

Claim: In the Snowshoe elliptic curve and in the integer ring modulo q, $z = a + h \cdot b$ is as indistinguishable from random as an ideal PRF(a || b).

Proof:

The value h is guaranteed to be uniformly randomly chosen. Define h = PRF(u || v). The client provides u,a, and the server provides v,b. Either the client or the server is guaranteed to be legitimate, and either one can cause PRF(u || v) to be random by submitting truly random input.

In reality the client is able to force some bits of h to selected values through exhaustive search of the input u. To prevent this from becoming an attack it is also important to note that $b = x \cdot v$, and x is a secret value of the server. Therefore h can only be selected blindly by the client and an attack attempting to create a collision with a specific value can only succeed with the probability of $\frac{1}{q-1}$ since different random values for x,y are chosen for each instance of the protocol.

The third modification is due to PAKE2+. The other modifications are introduced by Tabby PAKE in order to (first, second) close the backdoor presented by N, M in SPAKE2, and (fourth) improve efficiency by 25% during evaluation of the simultaneous scalar multiplication.

3.4.2 Forward and known-session secrecy

The Tabby PAKE protocol provides forward secrecy, meaning that compromise of the password or shared secret key K does not reveal the shared secret key K for any other previous session. The secret key [K(64 bytes)] = H(E || X' || Y' || Z || ExtraTags). The value E is deterministically based on the password, but the other points that are hashed vary for each instance of the protocol:

Server $X' = x \cdot G + E$

Client $Y' = y \cdot G + E$

Server $Z = x \cdot Y + h \cdot x \cdot V = (x \cdot y + v \cdot h \cdot x)G$

Client $Z = (y + v \cdot h)X = (x \cdot y + v \cdot h \cdot x)G$

where h = H(X' || Y'), (v, V) = GeneratePasswordVerifier(username, realm, password, salt)

The uniform random values x,y are unique to each instance and are chosen by each side independently to guarantee that the session secret will be unique for each instance even if one side purposefully reuses a random value. Since scalar elliptic curve point multiplication as a function is a bijection between x,y and the points X',Y', there is no aliasing of multiple values x,y to the same secret point Z. There is some small chance, however, that the hash function H will map two different Z points to the same key. Since H is treated as a PRF this effect can be ignored in the random oracle model.

In the random oracle model, knowledge of the secret key K does not reveal point Z without exhaustive search of all points, which is infeasible. If the secret key K is revealed, then the attacker does not gain any other information that is useful. This is the most likely type of practical compromise, since the secret point Z is computed and then quickly erased from stack memory. Therefore, even if the value of K is revealed (the secret key) for one session, no other sessions are compromised. **This guarantees known and forward secrecy of other sessions on the reveal of secret key K of one session.**

If H is not a true PRF despite its good design, then reveal of the secret key K implies reveal of the secret point Z. Taking this as a starting point:

Claim: The repeated reveal of Z does not also reveal the values x,y,v,V for any session.

Proof:

Rearranging the equation: $Z = x(Y + h \cdot V) = x(y + h \cdot v)G = w \cdot G$

In the expression $y + h \cdot v$, h is known to an attacker.

In the least, the value $x \cdot y \cdot G$ is unknown to the attacker and can take on any value in the q-order ring of the elliptic curve.

This effectively "encrypts" the other term $x \cdot v \cdot h \cdot G$. And similar to an ideal cipher, an attacker cannot gain any advantage above random chance at guessing the "encrypted" value $x \cdot v \cdot h \cdot G$.

By the attacker may effectively attempt some sort of "chosen-plaintext attack."

The attacker does know the value h, and v is constant: $x \cdot v \cdot h \cdot G = x \cdot A$, where the attacker may have some knowledge of A through repeated queries.

However, this term is further protected by the multiplication of x for each session. Therefore the value v is protected by the CDH assumption.

Therefore, no information is gained by the attacker by repeated reveals of Z.

The values x,y for each session are considered ideal independent random variables. As a result revealing them does not give an attacker any information about the value of x,y for other sessions. However, revealing one of these values and the secret key K is devastating if the hash is not a true PRF:

Claim: Revealing x,Z or y,Z allows for recovery of the password.

Proof:

Revealing x, Z and public data h, Y' allows $Z = x \cdot Y + h \cdot x \cdot V$ to be rearranged: $Z = x \cdot ((Y' - E) + h \cdot V)$

Computing A = $[x^{-1} (\mathrm{mod}\, q)] \cdot Z$ yields $A = Y' - E + h \cdot V$.

And the Y' term can be stripped: $h \cdot V - E$.

This expression allows for polynomial-time exhaustive search for the password, violating the claimed security level. A similar attack will break the protocol if y, Z are revealed.

As shown above, the shared secret point Z is linearly related to the point $x \cdot y \cdot G$ (as in EC-DH) on both the client and the server side. Without knowledge of (x, Y), (X, y), or (x, y) it is infeasible to reproduce the shared secret point Z and therefore the secret key K for a session. Therefore, even if the value of v is revealed (the password is revealed), previous sessions are not compromised. **This guarantees forward secrecy of prior sessions on the reveal of the password.**

Two claims have been made to arrive at this conclusion:

Claim: The reveal of V does not also reveal the value $Z = x \cdot Y + h \cdot x \cdot V = (x \cdot y + v \cdot h \cdot x)G$ unless the

private key x is known.

Proof:

Note that knowledge of either v or V implies knowledge of both, because V can be generated from v, and because V can be used to find v in polynomial-time.

Rearranging the equation: $Z = x \cdot (Y + h \cdot V)$

The values Y,h are fully determined by the knowledge of V.

So the equation can be further simplified: $Z = x \cdot A$, where A is known by the attacker.

The value of Z in this case is protected by the CDH assumption, and therefore the forward secrecy of Z is proven by reduction to this hard ECDLP problem.

Claim: The reveal of v does not also reveal the value $Z = (y + v \cdot h)X = (x \cdot y + v \cdot h \cdot x)G$ unless the private key y is known.

Proof:

Similarly rearranging the equation: $Z = yX + A$, where A is known by the attacker.

The attacker can thus reduce the problem to finding $Z' = y \cdot X$, where X is a point known by the attacker.

The value of Z in this case is also protected by the CDH assumption, and therefore the forward secrecy of Z is proven by reduction to this hard ECDLP problem.

Note that these proofs do not require H to be a true PRF.

After the password is revealed, it is not true that session data will remain secret. With knowledge of V, an attacker can masquerade as the legitimate server at that point. This is why two-factor authentication is important and the server should also be authenticated by other means e.g. PKI as is the current standard practice.

3.4.3 Augmented property

The server is only ever given V = vG during account creation. To directly recover the value v used by the client to log in, the strong ECDLP problem must be solved. Therefore the best attack for the server (or a hacker who has recovered the account database) is instead to brute-force the password. It is normal for password cracking to be the best option in this case, so no usual security guarantees are lost.

3.4.4 Backdoor-free

The Tabby PAKE code itself has no arbitrary constants and only implements the protocol as specified above.

The elliptic curve group chosen for Tabby follows the SafeCurve guidelines and as a result has a rigid specification in which the author cannot imagine a way to hide a backdoor.

Tabby is also based on the Cymric random number generator, which has a robust and simple design, including a wide-pipe key generation step, a second one-way function (ChaCha) for actual generation, and extra entropy mixing. The output of Cymric is used safely as scalars for elliptic curve multiplication in the Tabby PAKE protocol to avoid the impact of misuse of the generator. The Tabby PAKE protocol never directly reveals random data to active or passive attackers. In short it seems impossible to hide a backdoor in the random number generator.

The hash function H, defined as BLAKE2b, has many selected constants but has been shown by its designers to resist differential cryptanalysis and all other known attacks.

The PBKDF function, defined temporarily as Lyra, reuses the BLAKE2 functions and so does not appear to have any opportunity to introduce a backdoor.

3.5 Performance Measurements

Benchmarks have been conducted on the Intel Haswell architecture, which has been commonly available in personal computers since mid-2013. This is expected to provide good examples of performance on both commodity server hardware and modern desktops.

The performance on ARM-based platforms is not exceptional due to the lack of optimization. These results are excluded because mobile devices are expected to predominantly take on the Client role in the protocol. In the Client role, the execution time is overwhelmingly dominated by the PBKDF algorithm and so compatibility and simplicity is the more important goal on mobile platforms.

3.5.1 Methodology

On Sandy Bridge and newer processors, the RDTSC instruction does not return actual clock cycle counts. Instead it returns a clock at a fixed frequency, which is determined during testing. The actual processor cycles are typically at a higher Turbo Boost frequency under load, which varies unpredictably.

Even when Turbo Boost is disabled, the RDTSC clock is sometimes at a different rate than the CPU clock rate, so on some machines a correction factor is needed in the results.

To measure the cycle count for an operation, Turbo Boost was manually disabled on each platform. The median of 10,000 measurements was taken, and repeat measurements indicate that they are accurate to ~100 cycles. Following this methodology, the resulting cycle counts on all test machines of the same architecture (Sandy Bridge, Ivy Bridge, Haswell, etc) with varying clock rates match, which serves to validate the approach.

To measure the average wall-time for an operation, the median of 10,000 measurements was taken. While testing the machine was not involved in any other processing.

3.5.2 Baseline

The DragonSRP[15] project is based on OpenSSL and provides 1024, 2048, and other discrete logarithm size options for security. Its convenient benchmark tool was compiled with -O3 on the same machine as the Tabby PAKE software. After reading through the code and some probing it looks like nearly all of the execution time is in the OpenSSL routines as expected.

The server operation execution time is also compared with the elliptic curve Diffie-Hellman (EC-DH) operation on the same curve. Ideally, Tabby PAKE server operations would take about the same time as EC-DH so that it can be just as widely deployed as normal PKI-based handshakes.

3.5.3 Results

Benchmarks on Macbook Pro (2.4 GHz Core i5-4258U Haswell, late 2013):

RDTSC instruction runs at 2.39992 GHz. No scaling is needed.

OpenSSL SRP benchmarks:

- 1024-bit logarithm performance: `**2742**` avg usec per operation
- 2048-bit logarithm performance: `**8384**` avg usec per operation

Tabby PAKE benchmarks:

- Client generated server verifier for password database in `**90994492**` cycles, `**31653**` usec (one sample)
- Server password challenge generated in: `**77184**` median cycles, `**26.7673**` avg usec
- Client proof of password generated in `**89073576**` cycles, `**30908**` usec (one sample)
- Server proof of password generated in: `**172992**` median cycles, `**59.8146**` avg usec
- Client checked server password proof in `**540**` cycles, `**1**` usec (one sample)

Other Tabby Operations:

- Created a new server key pair in `**352724**` cycles, `**128**` usec (one sample)
- Tabby sign: `**57316**` median cycles, `**20.16**` avg usec
- Tabby verify signature: `**136516**` median cycles, `**48.0554**` avg usec
- Tabby reject signature: `**136776**` median cycles, `**48.0778**` avg usec
- Generated a client key in `**172120**` cycles, `**60**` usec (one sample)
- Periodic server rekey in `**178468**` cycles, `**65**` usec
- Tabby client rekey: `**58000**` median cycles, `**20.1884**` avg usec
- Tabby server EC-DHE handshake: `**121896**` median cycles, `**42.3002**` avg usec (`**23640**` connections/second)
- Tabby client EC-DHE handshake: `**171412**` median cycles, `**59.5031**` avg usec

3.5.4 Discussion

The Tabby server EC-DHE handshake measurement is essentially the time it takes to do a single EC-DH (variable-base point multiplication) operation. It takes about 122,000 Haswell cycles on the Snowshoe curve. This is the limiting operation for accepting new connections secured by PKI-supplied authentication, as is done in TLS.

For Tabby PAKE, the first round takes just 78,000 cycles on the server, so it is actually faster than EC-DH. The second round takes 173,000 cycles, just 40% slower than EC-DH. On this laptop between the two operations, the server can validate **11,500** connections/sec using password authentication. This is about half the speed of EC-DH on the curve, so a good rule of thumb is that Tabby PAKE is half as fast as EC-DH.

In comparison, SRP with a 1024-bit discrete log problem can handle **365** connections/sec, and SRP with a 2048-bit discrete log problem can handle **120** connections/sec.

According to the ECRYPT II Recommendations[16] from 2012, it is notable that the security level of 2048-bit discrete logarithms is considered much lower than Tabby PAKE, roughly “102 bits” interpolating between the table entries on the website. And 1024-bit discrete logarithms (the default often used for SRP) only offer about “75 bits” of security.

In summary, Tabby PAKE server software is able to handle logins 100x faster than similar server software yet at a higher security level than previously possible.

4 Software Architecture

4.1 Overview

In order to give a full proof of security for Tabby, a concrete specification that meets the stated security goals is also provided in this document, corresponding to the publicly available software that implements the Tabby protocol.

The overall design of Tabby is simple and does not attempt to do anything new or unusual. It simply puts together a best-of-breed set of existing algorithms in well-studied ways to arrive at an efficient and secure overall system.

The Tabby reference software is architected from several components:

- **Cymric**[17]: Cross-platform Cryptographically-Secure Pseudo-Random Number Generator (CSPRNG) developed for Tabby.
- **Snowshoe**[18]: Portable, secure, and fast elliptic curve math library developed for Tabby.
- **BLAKE2**[19]: Strong PRF with 64 byte output and high speed designed by Aumasson et al in 2013.
- **Lyra**[20]: Password-based key derivation function designed by Almeida et al in 2014. *This component can be easily switched out for another PHC candidate based on what happens in the competition. Lyra was chosen somewhat on a whim and for purposes of the security proof is treated as a generic ideal construction of a password-based key derivation function. The author has no affiliation with the developers of Lyra.*

This document will describe the constructions of Cymric and Snowshoe, which attempt to provide ideal primitives. Non-idealities are intended to be fully explored in this document and are incorporated into the security proof.

The BLAKE2 and Lyra components were developed by other designers and are treated more generically by this document as ideal PRF and PBKDF primitives, respectively. It is also trivial to replace either of these components with more vetted options for practical systems based on the Tabby protocol.

4.2 Random Number Generator: Cymric

For modern applications it is important to consider the use of Tabby on thin client mobile devices as well as servers typically found in computing clusters. Both of these environments have limited access to entropy, and so a suitable random number generator must be chosen. Speed is paramount for battery life and interactivity in the mobile case, and speed is important for serving many clients and avoiding denial-of-service attacks in the server case.

Random numbers are an integral component for each round of the Tabby protocol. They provide the forward secrecy property, generate the password salt, and ensure that the zero-knowledge proof does not critically fail.

The goal of the Cymric pseudo-random number generator is to produce output that is unguessable to an adversary with limited access to the system. To achieve this goal, Cymric accesses sources of entropy on the system and attempts to gain at least 256 bits of entropy. Entropy is defined here as seed bits that cannot be guessed by an attacker. After roughly 256 bits of entropy it is assumed that it will be easier to attack other parts of the Tabby design rather than the random number generator.

Any data used to generate a random number is securely erased from stack and heap memory shortly after use to avoid it being leaked somehow later.

The internal state of the generator is never be directly exposed in the output, and a certain amount of the internal state is held in reserve and not directly used to produce the output, to make the next iteration of the state more difficult to guess even after a compromise of the previous generator state.

A small amount of new entropy is mixed in with each new generated number without affecting the speed of the generator nor requiring an expensive background thread that may hurt the battery life of mobile devices or scalability on servers.

A secondary goal of Cymric is to run quickly so that users of the overall software (Tabby) do not disable the password security in order to “fix” performance problems. For this reason, Cymric never blocks for entropy collection. Specifically, when collecting entropy from /dev/random Cymric will read as much as it can without blocking, and read the remaining required bytes from /dev/urandom. It then becomes the responsibility of the system administrator to ensure that enough entropy is present in the system to safely run cryptographic software.

Cymric may be the weakest link in the security chain if the system it is running on does not have enough entropy available in the Operating System entropy pool. For example, if Tabby is running on an embedded Linux device and the /dev/random pool is entirely guessable by an attacker, then the best attack against Tabby would be to exhaustively search for the internal state of the Cymric generator, checking against observed output, which may not meet the “125 bit” security goal. These attacks are well known and may be used to attack almost any cryptographic system. Cymric does not introduce any new defense against this class of attacks apart from mixing additional sources.

Cymric has a simple self-check to verify that in the platform it is compiled for, there are a minimum number of entropy sources available.

4.2.1 Entropy Sources

Any of these entropy sources are used when they are available:

- Uninitialized memory from input state object
- /dev/random (Attempts to read 256 bits)
- /dev/urandom (Reads remaining 256 - bits from here)
- Two samples of the processor cycle counters
- Microsecond-resolution timestamp
- arc4random_buf()
- getpid()
- pthread_self()
- gettid() / syscall(__NR_gettid)
- win32 GetCurrentThreadId()

- win32 CryptoAPI CryptGenRandom (256 bits)
- win32 GlobalMemoryStatus MEMORYSTATUS

And some extra data to make attacks against the generator more difficult:

- Atomically incrementing 32-bit counter for each seed/generate
- stdlib srandom(time(0)) and random()x2
- stdlib srand(time(0)) and rand()x2

When at least 425 bits of data overall are collected, seeding succeeds. These sources are mixed together with the BLAKE2 hash function into the 64 byte state. Note this does not mean 425 bits of entropy, but is used instead as a self-test to verify proper function. It is assumed that the operating system will provide 256 bits of entropy and the rest are used to strengthen that guarantee.

The scratch buffer is securely erased from memory after seeding to avoid leaking any of the input source data.

4.2.2 Generator

The generator always validates its input and verifies that the state is seeded.

The input to the generator is a 64 byte state buffer. The low 32 bytes are treated as a 256-bit secret key, and the next 8 bytes are treated as a secret 64-bit IV for the ChaCha[21] cipher function with 20 rounds.

Opportunistically, before initializing the cipher a few fast entropy sources are mixed into the IV when they are available:

- Processor cycle counter is added into the low 32 bits of the IV.
- Current thread ID is added into the high 32 bits of the IV.
- Atomic 32-bit incrementing counter is added into the low 32 bits of the IV.

Note that these modifications are also directly to the 64 byte state.

Then the ChaCha function "encrypts" the output random number buffer, which securely mixes in any uninitialized data in the buffer.

After encryption completes, the processor cycle counter is added into the high 32 bits of the IV part of the 64 byte state.

Finally, the 64 byte state is erased by hashing it with the BLAKE2 hash function. The output is the new 64 byte state. Note that the high 24 bytes are never used to generate random numbers, meaning that if one instance of the ChaCha function is fully compromised, there will still be 192 bits of unknown state going into generating the next random number. This should be impossible in the first place, but Cymric does fail gracefully.

4.3 Safe Software Design Practices

Cymric, Snowshoe, and Tabby are built with the same solid set of safe software design practices.

Cryptographic software is exceptionally hard to write because it does not fit the standard test-driven development methods practiced commonly by software developers.

Innovation is usually frowned upon for cryptographic code. None of the ideas presented here are new. The goal was to faithfully reproduce the studied results of other researchers.

In general writing less code is better. Trading speed for simplicity is always helpful. Code reuse helps make bugs more visible. Where code cannot be easily reused as a function, macros are a good substitute to avoid copy/pasting and provide self-documentation. Runtime self-tests catch many types of misuse and unforeseen environment-specific problems. Input checking catches many more types of misuse. APIs can be designed to discourage misuse.

Usage of architecture-specific assembly code is minimized in favor of portable C code to encourage code reuse between architectures, which improves the visibility of bugs and makes it easier to achieve code-complete testing.

Clean, easy to read code is easier to audit. Reading through old code every few months has been a way to add

more confidence. Testing code on many different platforms with different compilers and as many different debugging tools and options as possible helps to motivate eliminating all the obvious problems.

The code is built from small components with well-defined minimal interfaces that are individually tested, well-documented and audited, similar to the microkernel philosophy.

The APIs each have an initialization function that verifies that the basic assumptions of the library are not violated and that the library passes basic self-tests. Other functions verify that initialization has occurred and fail with an error if the initialization step is skipped.

Usage of the heap is minimized and well defined as it is acknowledged that data on the stack is much more likely to be erased securely in light of the disk-paging system of modern operating systems.

The APIs involve as few functions as possible with unambiguous arguments and full documentation including preconditions. All C functions follow the convention that zero is a successful return value and any other return value is a failure result, making it difficult to misinterpret a return value. All inputs are validated at runtime even in release mode. State data passed between functions all have canary bytes to check for uninitialized states to avoid misuse.

4.4 Elliptic Curve Group: Snowshoe

The elliptic curve group used by Tabby has an explicit practical construction called Snowshoe. Snowshoe was selected to fulfill the high-speed goals of Tabby. Alternative elliptic curve groups are definitely usable and superior in some ways. Tabby specifies one SafeCurve group to guarantee it will not be misused. To adapt Tabby to other suitable groups it should be enough to demonstrate that the other groups offer the same security properties as this group, and the same security proof will cover the new construction.

4.4.1 Overview

The goal of Snowshoe is to enable secure elliptic curve group operations:

- Multiplication of a variable scalar by the fixed generator base point: **R=[4]kG**
  - *This is used for key generation and signing.*
- *Multiplication of a variable scalar by a variable base point:* ***R=4kP***
  - *This is used for Diffie-Hellman key agreement.*
- *Sum of simultaneous mixed-base multiplication:* ***R = 4aP+4bG***
  - *This is used to verify signatures.*
- *Sum of simultaneous variable-base multiplication:* ***R=4aP+4bQ***
  - *This is a rare feature useful for faster forward secrecy in Diffie-Hellman key agreement and also used to accelerate Tabby PAKE.*
- ***Elligator point decoding****, described later and used for Tabby PAKE.*
- *Other usual operations: Addition, negation, input validation, and big integer arithmetic modulo the group order "q."*

The multiplications by 4 above are multiplications by the cofactor of the curve to avoid small-subgroup attacks.

Curve specification:

- "125 bits" of security
- Field math: F_p^2 modulo $p = 2^{127} - 1$ (familiar grade school complex math)
- Twisted Edwards GLS curve: $E = a \cdot u \cdot x^2 + y^2 = 1 + d \cdot u \cdot x^2 \cdot y^2$
- $u = 2 + i$, $i = \sqrt{-1}$, a = -1, d = 109
- Group size: #E = 4q, indicating a cofactor of 4
- q is a large 252-bit prime =

0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFA6261414C0DC87D3CE9B68E3B09E01A5

Compatible with SafeCurve[22] guidelines:

- Base point and its endomorphism are of order q
- Resists Pollard rho method with a large prime group order q
- Not anomalous, and embedding degree is large
- CM field discriminant is large
- Fully rigid curve design: All parameters are chosen for simplicity
- The curve supports constant-time multiplication with Montgomery Ladders

- Prevents small-subgroup attacks by multiplying by the cofactor 4
- Prevents invalid-curve attacks by validating input points
- The doubling and unified addition formulae used are complete
- Supports Elligator construction, though only decoding is implemented

Implementation features:

- Uses constant-time arithmetic, group laws, and point multiplication
- Critical math routines have proofs of correctness
- Designed for testability:
  - 100% code coverage for math routines
  - Power-on self test
  - Full input checking

Performance features:

- Fast, simple extension field arithmetic: F_p^2 modulo $p = 2^{127} - 1$
- Fast, simple group laws: Extended Twisted Edwards
- Fast endomorphism-accelerated scalar multiplication using GLV-SAC
- Fastest public domain software for elliptic curve math

SUPERCOP Level 0 copyright/patent protection:

There are no known present or future claims by a copyright/patent holder that the distribution of this software infringes the copyright/patent. In particular, the author of the software is not making such claims and does not intend to make such claims.

4.4.2 The Curve

Recently enough research has been done to prove that using curves equipped with endomorphisms is a safe practice. Until now there have been no public domain implementations of curves with efficient endomorphisms that run in constant-time despite the clear advantages in speed. Snowshoe uses a curve equipped with an efficient endomorphism described below.

Recently Bos, Costello and Miele[23] analyzed the practical security of the BN254 curve and considered other similar curves with endomorphisms that are automorphisms. The practical speedup for the Pollard Rho technique to solve the ECDLP due to automorphisms is roughly 2x. The GLS endomorphism is an automorphism on the curve, meaning only about 1 bit of security is lost by using the endomorphism. This encouraging result directly motivated the formalization of the Snowshoe curve.

Snowshoe represents points in (X,Y) form rather than compressing to X-only coordinates for efficiency and simplicity since the curve endomorphism is used.

The selected curve equation is not original, but the work to produce it was independently arrived at by the author to verify that the parameters are suitably “rigid” enough to not indicate the presence of a backdoor.

4.4.2.1 Background on EC Endomorphisms

Elliptic curve endomorphisms are defined as functions that operate on a curve point and yield a new curve point with the relationship:

$P_2 = endo(P_1) = k \cdot P_1$, where k is ideally a large constant.

Essentially, this enables point multiplication by a constant practically for free, where normally this would be a very expensive operation.

On curves with 2-dimensional endomorphisms, there is only one such function. However, on curves with higher-dimensional endomorphisms there are more functions that each can quickly multiply by different large constants.

Trying to go for a 4-dimensional GLV-GLS[24] combination is possible but seems like it adds more complexity than is comfortable. The GLV endomorphisms put restrictions on the base field that would not allow the pleasant base field math used by Snowshoe. $X_2 = 2 \cdot X \cdot Y \cdot (2 \cdot Z^2 - Y^2 + u \cdot X^2)$ There are also not a lot of good examples of 4-dimensional endomorphisms in the literature and less analysis.

The GLS[25] endomorphism alone, however, only requires an extension field and as a result is incredibly easy to implement. One of the main goals of Snowshoe was simplicity to make it easy to audit, so this was immediately attractive.

Last year Q-Curves[26] equipped with efficient 2-dimensional endomorphisms through the CM method were introduced. The problem with Q-Curves is that they require the curve equation to include much larger coefficients, which will slow down evaluation. The advantage of Q-Curves is that the existence of the endomorphism does not help to make the ECDLP any easier, while the GLS endomorphism does reduce security level. This makes Q-Curves more interesting for accelerating very high security levels above 128 bits. Also interesting is that Q-Curves with endomorphisms cannot meet all the requirements for the SafeCurve checklist, which is questionable because this variety seems to be considered secure.

4.4.2.2 Coordinates: X,Y versus X-only

Another recent trend has been to perform all the operations on the X-line and representing points compressed as the X coordinate. This is seen in Curve25519, in experimental Q-curves[27], and in Hamburg's implementation[28]. The downside of X-only arithmetic is that it gets complicated when used with endomorphisms and it takes impressive experience in the related mathematics to combine the two in a safe way.

According to the group laws presented in EFD[29] and the results from Costello et al, a 2-dimensional Montgomery scalar multiplication performs 1 add and 1 double-add per bit. Yet it has been shown[30] that twisted Edwards group laws allow for a protected table-based approach that requires 1 double and 0.5 adds per bit with a practical window size of 2 bits.

With Snowshoe's finite field, squares require 0.67x the time of a multiply, so the operation count is roughly 13.36 multiplies per bit for Montgomery curves, and (5.68 + 5.68 + 9) / 2 = 10.18 multiplies per bit for Edwards curves. Note that the estimate of 9 multiplies above is pessimistic.

This demonstrates that (X, Y) coordinates are always preferred for speed for all types of applications. Furthermore (X, Y) coordinates are much more flexible, as they are useful for signature verification. And this allows Snowshoe to offer **all** elliptic curve operations through a single API.

The size difference between 32 byte public keys (X-Only) and 64 byte public keys (X,Y) is inconsequential in nearly every case. RSA public keys are much larger and are unfortunately still used for many real applications, for example.

4.4.2.3 Design of Curve Equation

The Twisted Edwards form $E: a \cdot u \cdot x^2 + y^2 = 1 + d \cdot u \cdot x^2 \cdot y^2$ allows for efficient constant-time EC point addition and EC point doubling formulae with a minimal number of operations for points represented in extended Twisted Edwards (X,Y,T,Z) form as shown by the EFD.

Since multiplication by u appears in both the doubling and addition formulae, it was chosen to be as efficient as possible to multiply: $u = 2 + i$. This is the same as presented in the Ted1271gls[31] implementation by Longa. Another fast alternative would be to use $u = 1 + 2^{64} i$ but this requires practically the same operation count while adding more complexity. Nicer options such as "1+i" are unfortunately square in F_p^2 and will not work.

Sticking with the existing literature is best here.

Choosing a = -1 or 1 in the equation leads to known efficient EC point operation formulae, however a = -1 is more commonly found in the literature and was chosen.

The only remaining free parameter is d. Using a MAGMA[32] script (called magma_curvegen.txt in the source distribution) the following alternative curves were found that have maximum security for the given field, listed from smallest to largest:

```
     d=109 : #E = 4*q, q =
0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFA6261414C0DC87D3CE9B68E3B09E01A5

     d=139 : #E = 4*q, q =
0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFBE279B04A75463D09403332A27015D91

     d=191 : #E = 4*q, q =
0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF826EDB49112B894254575EA3A0C8BDC5

     d=1345: #E = 4*q, q =
0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF80490915366733181B4DC41442AAF491

     d=1438: #E = 4*q, q =
0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFC96E66D7F2A4B799044761AE30653065

     d=1799: #E = 4*q, q =
0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF8FF32A5C1ACEC774E308CDB3636F2311

     d=2076: #E = 4*q, q =
0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF81EBFEA8A9E1FB42ED4A6EBB16B24A91

     d=2172: #E = 4*q, q =
0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF819920B3F8F71CD85DD3F4242C1B0E11

     d=2303: #E = 4*q, q =
0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF9B3E69111FF31FA521F8B59CC48B4101

     d=2377: #E = 4*q, q =
0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF94B9FB29B4A87B1DAEFA7A69FC19FD11

     d=2433: #E = 4*q, q =
0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF8F4C87E0F8EB73ABCB41D9C4CF92FC41

...
```

Note that it takes very little time to search for these curves with modern point counting implementations and it is easy to reproduce these results.

Some exploration was also done to try to find a value with lower Hamming weight but nothing was easily found. The smallest d = 109 was selected, which happens to be the same used in Ted1271gls, benefiting further from being already explored in the literature.

These clear, rigid selection criterion preclude the existence of a backdoor in the final Snowshoe curve equation:

$$E = -(2+i)x^2 + y^2 = 1 + 109(2+i)x^2y^2$$

This curve is isomorphic to the Weierstrass curve equation:

$y^2 = x^3 + A \cdot x + B$, where A =

170141183460469231731687303715884104864*i + 127605887595351923798765477786913078648

And B =

850705917302346158658436518579420314 30*i + 170141183460469231731687303715884101830

4.4.2.4 Curve Properties

On this curve, (0, 1) is conveniently the identity element and (0, -1) is of order 2. The codebase rejects x = 0 input entirely as invalid, though it is possible for the identity element to occur during evaluation.

Point negation is defined in Affine coordinates as:

-(x, y) = (-x, y)

Point negation is defined in extended Twisted Edwards coordinates as:

$-(x, y, t, z) = (-x, y, -t, z),\ t = \frac{xy}{z}$

4.4.2.4.1 Generator Point

The generator point on a cryptographic elliptic curve is the base point used for key generation. For this reason it is also referred to as the "base point." For rigidity, the first twisted Edwards curve point of q torsion with a small, real X coordinate was chosen as the Base Point, shown here in affine coordinates:

$B_X = 15$

$B_Y =$

0x6E848B46758BA443DD9869FE923191B0 * i + 0x7869C919DD649B4C36D073DADE2014AB

Values of X from [0..14] are either not on the curve, or q(X, Y) = (0, 0).

The base point B of a secure elliptic curve must be in the q torsion group such that qB is the identity element.

Since the Snowshoe curve has cofactor 4, there are actually 4 different types of base points on the curve, shown here in Weierstrass coordinates:

- qP = (18i + 36, 0)
- qP =

(6804124518483755145717709703422605848 8*i+5104996913781584013725510334082902 3597, 0)

- qP =

(1020999382756316802745102066816580472 21*i+1190912143226533915944322003750 55082094, 0)

- qP = (0, 1)

And in twisted Edwards coordinates:

- qP = (0, 0)
- qP = (0, 1)

Note that for all of these, 4qP = (0, 1). And the endomorphism of the base point B will be in the q torsion group by construction.

The endomorphism of the generator point in twisted Edwards coordinates, $(B'_X, B'_Y) = endo(B_X, B_Y)$:

$B'_X =$

0xA7B74573CBFDEDC58A1315F74055A23 * i + 0x453DBA2B9E5FEF6E2C5098AFBA02AD11

$B'_Y =$

0x117B74B98A745BBC226796016DCE6E4F * i + 0x7869C919DD649B4C36D073DADE2014AB

Using unit testing software the base point and the endomorphism of the base point were validated to be on the curve and verified to be of q torsion. See also the MAGMA script magma_generator.txt in the software distribution for more details.

4.4.2.4.2 Handling of Exceptional Inputs

Snowshoe's EC point multiplication routines are protected from some types of exceptional input. All of the input points are verified to exist on the curve. And points with x = 0 are entirely rejected.

The remaining points that are allowed to run through the math routines may not be order q. Recall that about 3/4ths of the points on the curve are of order 4q rather than of order q. Despite the use of efficient endomorphisms during point multiplication, these exceptional points are handled properly, and have the same results as a simple double-and-add reference multiplier. This has also been demonstrated experimentally by the unit tester.

The scalar inputs to the EC point multiplication routines can also taken on exceptional values. Variable-base and signature validation routines can accept scalar inputs that are zero, which will result in the identity element. However, the other EC point multiplication routines will produce invalid output when the scalar is zero.

To address this problem, the input scalars for all of the Snowshoe multiplication functions are checked to ensure they are in the expected range **0 < k < q**. This check is also done in constant-time. After writing the Tabby library, there does not appear to be any reason to allow for **k = 0** nor **k ≥ q**.

4.4.2.5 GLS Endomorphism

In Twisted Edwards coordinates, the endomorphism of the Snowshoe curve is:

$$endo(x,y) = (w'_x \cdot \bar{x}, \bar{y}) = \lambda(x,y)$$

$$u = 2 + i$$

$$w'_x = \sqrt{\frac{u^p}{u}} =$$

689853595270286368735396082714597189 31*i + 119563271493748934302613455993671912329

= 0x33E618D29DA66430D2B569B107BC1713 * i + 0x59F30C694ED33218695AB4D883DE0B89

In the GLS method, $\lambda = \sqrt{-1} (\mathrm{mod}\, q) = \sqrt{q-1} (\mathrm{mod}\, q) =$

667526209023283335426145907808145682639669420444541460451714799617543798 5167

= 0xEC2108006820E1AB0A9480CCBB42BE2A827C49CDE94F5CCCBF95D17BD8CF58F

The endomorphism is defined in Weierstrass coordinates as:

$$endo(x,y) = (u \cdot \frac{x^p}{u^p}, \sqrt{u^3} \cdot \frac{y^p}{\sqrt{u^{3p}}}) = \lambda(x,y)$$

To simplify recall that $x^p = \bar{x} = conj(x)$ over F_p^2.

Therefore:

$$w_x = \frac{u}{u^p} =$$

102084710076281539039012382229530463437*i + 340282366920938463463374607431768211 46

$$w_y = \sqrt{\frac{u^3}{u^{3p}}} =$$

818535452895934200673828431406455860 77*i + 145299643018878500690299925732043646621

And finally: $endo(x,y) = (w_x \cdot \bar{x}, w_y \cdot \bar{y}) = \lambda(x,y)$

It has been verified that $\lambda = \frac{p-1}{Tr(E)} (\mathrm{mod}\, q)$, and that $\lambda P = endo(P)$ for various P.

4.4.2.5.1 Scalar Decomposition

The intent of scalar decomposition is to decompose the scalar k into two sub-scalars k_1, k_2 such that $k = k_1 + k_2 \cdot \lambda$ for some constant λ and the length of each sub-scalar in bits is the same. This must be done in constant-time since the value of k is supposed to remain secret in most cases.

After the decomposition is complete, EC point multiplication may be evaluated as:

$$Q = \lambda P = endo(P)$$
$$kP = k_1 P + k_2 Q$$

where k_1, k_2 are each half the size of k.

The simultaneous evaluation of $k_1 P$ and $k_2 Q$ turns out to effectively halve the number of EC point double operations required to evaluate kP while keeping the same overall number of EC point addition operations.

The values k_1, k_2 are signed and can each be negative. So a fast constant-time conditional point negation must be performed on the base points before scalar multiplication. An alternative approach was taken in Costello's X-line Endomorphisms paper to offset the lattice decomposition to avoid negative values, but it is actually faster and easier to do conditional negation.

Lattice decomposition with precomputed basis vectors is the usual approach to decompose the scalar. Suitable short basis vectors have been provided by Smith[33] and they were scaled to work modulo q rather than 4q.

Given:

Base field modulus $p = 2^{127} - 1$

Curve order r = 4q

Group order q =

723700557733226221397318656304299424070994123655496019766597502163450055 9269

Group order trace t = 14241963124919847500

λ =

667526209023283335426145907808145682639669420444541460451714799617543798 5167

The orthogonal basis vectors due to Smith are:

b1 vector $\langle x_1, y_1 \rangle = \left\langle \frac{p-1}{2}, \frac{-t}{2} \right\rangle$

b2 vector $\langle x_2, y_2 \rangle = \left\langle \frac{-t}{2}, -\frac{p-1}{2} \right\rangle$

Principal constants are:

qround = q div 2 + 1 =

0x7FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFD3130A0A606E43E9E74DB471D84F00D3

$$A = floor(\frac{p-1}{2}) = 2^{126} - 1$$

$$B = floor(\frac{t}{2}) = 0x62D2CF00A287A526$$

Multiplication by A is replaced with shifts and subtractions in software.

B is also a relatively small 63-bit constant.

Solving for k_1, k_2 using 256-bit big integer math:

Step 1. $a_1 = A \cdot k \quad a_2 = B \cdot k$

Step 2. $z_1 = floor(\frac{a_1 + q_{round}}{q}) \quad z_2 = floor(\frac{a_2 + q_{round}}{q})$

Step 3.
$$k_1 = k - (z_1 \cdot A + z_2 \cdot B)$$
$$k_2 = z_1 \cdot B - z_2 \cdot A$$

This decomposition guarantees that k_1, k_2 are 126 bits or less. As shown by Smith, $\left|\left|k_i\right|\right| \leq \left|\left|\frac{p-1}{2}\right|\right| = 2^{126} - 1$, so their magnitude can be represented in 126 bits. The division by 2 is due to scaling the basis vectors as suggested by that paper to work over q modulus rather than 4q modulus.

Since the decomposition works modulo q instead of modulo 4q, the base point(s) must be premultiplied by 4 before scalar multiplication proceeds to avoid small-subgroup attack. This needs to be done anyway and is equivalent to clearing the low 2 bits of the scalar, except that it is more efficient to reduce the sub-scalar length by 1 bit and premultiply by 2 doubles in the case of variable-base scalar multiplication than it is to use the unscaled bases and perform an extra EC point addition during evaluation.

See magma_decompose.txt in the source distribution for more information and a MAGMA script that implements the decomposition.

4.4.2.5.2 Constant-Time BigInt Division

The divisions in step 2 of the Scalar Decomposition algorithm above are tricky to implement in constant-time. The inputs are unreduced 512-bit products. The technique from section 4 of a paper by Granlund and Montgomery[34] is used, which seems to be the most efficient constant-time approach and fairly straight-forward.

Algorithm for Unsigned Division:

To compute $\mathrm{r} = \mathrm{floor}(\frac{\mathrm{p}}{q})$ in constant time, let $q < 2^L \quad p < 2^N$.

L, N are some number of bits. Usually on the order of 252 and 512 respectively.

The value of unsigned integer $m' = \frac{2^N(2^L - d)}{d} + 1 = \frac{2^{N+L}}{d} - 2^N + 1$ is precomputed.

Step 1. $t = (m' \cdot p) \gg N$

Step 2. $\mathrm{s} = \mathrm{t} + ((\mathrm{p} - \mathrm{t}) \gg 1)$

Step 3. $r = s \gg (L - 1)$

This seems simple but it actually turns into about 200 lines of C code mainly due to the large size (512 bits) of the m' and p values in step 1.

This method is more efficient than naive approaches because the value of m' is roughly 126 bits smaller than it would be without step 2.

The runtime of this algorithm does not affect the performance of Tabby and is used in several places, for scalar decomposition and to reduce random numbers modulo q to produce elliptic curve private keys, as examples.

This code can be found in **misc.inc** of the software distribution. C macros are used to reduce the amount of copy/pasted code. Also see **magma_unsigned_remainder.txt** for more details.

4.4.3 Snowshoe Software Architecture

Snowshoe is built from simple modular layers, each building on the previous layer. The advantage of this nested construction is that each layer can be independently unit tested, audited, and more easily replaced in the future:

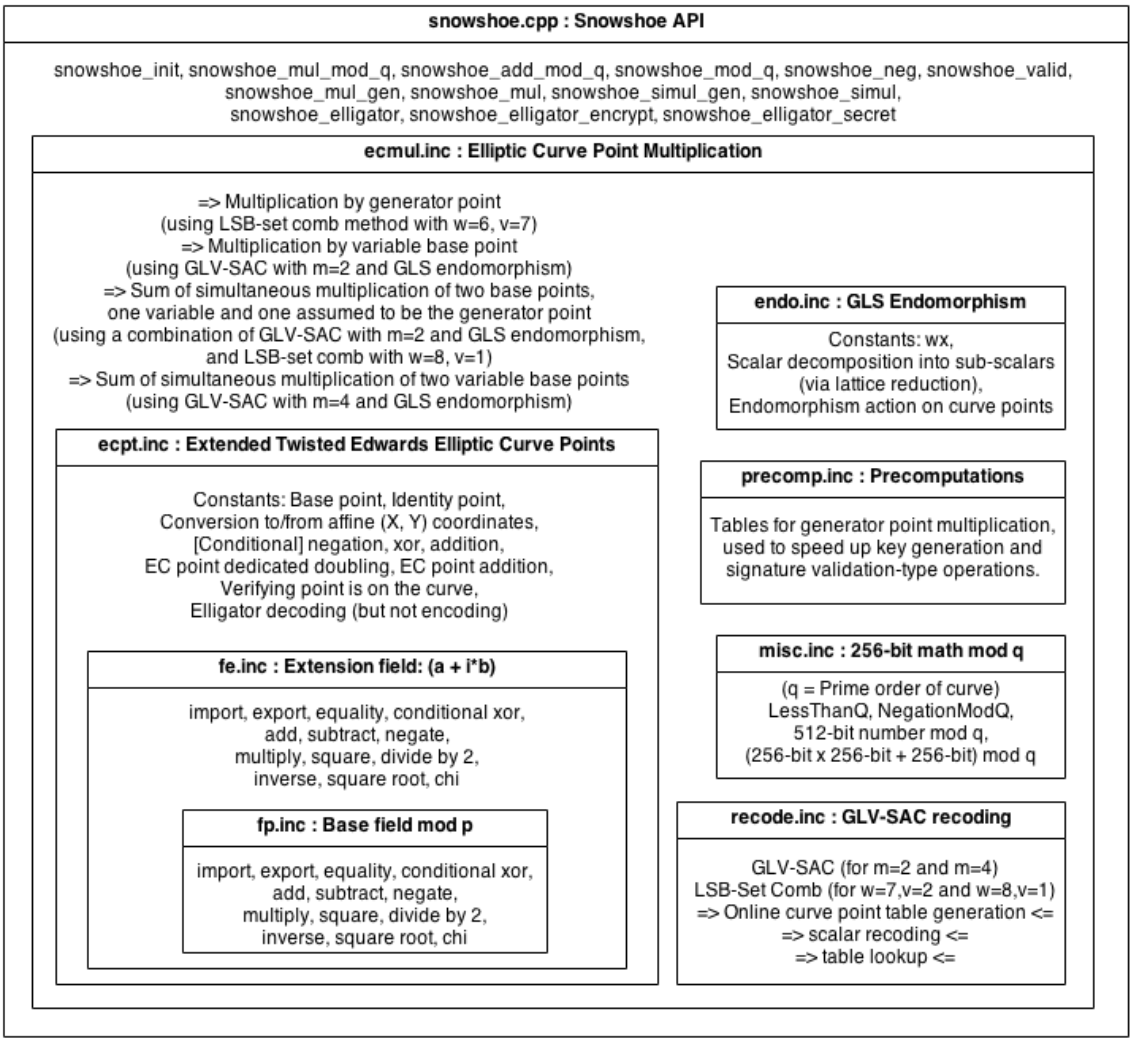

Fig1: Map of the Snowshoe elliptic curve math library

The Snowshoe elliptic curve group is conservative, efficient and flexible. Math on Snowshoe is significantly faster than the Curve25519[35] and Ed25519 curves, yet slower than experimental approaches based on Kummer surfaces[36] or 4-dimensional GLV[37]. It makes a comfortable trade-off between complexity and performance. In the future using Kummer surfaces will likely be the best option as these allow for EC point multiplication that is roughly two times faster than math on Snowshoe without any obvious disadvantages.

4.4.4 Constant-Time Operations

There are two major types of variable-time operations that need to run in constant time for EC point multiplication:

- Conditional operations
- Table lookups

Conditional operations

The constant-time conditional selection primitive used throughout Snowshoe is based on masking. A mask is a 64-bit value that has all bits set to 1 or all bits set to 0. A 1 bit value can be used to construct a mask by negating the value and sign-extending it to 64 bits. Then between two inputs X, Y, the output R can be selected based on the mask:

$$R = X \oplus ((X \oplus Y) \,\&\, mask)$$

This only requires 3 operations and processes 64 bits at a time. The values of X, Y are commonly larger than 64 bits. Using fast vector operations or a loop this selection primitive is applied to the entire size of the inputs.

Any conditional operations can then be implemented by unconditionally performing that operation, and then conditionally selecting its result.

Table lookups

Constant-time table lookups can be implemented more efficiently in terms of operation density than conditional operations:

Step 1. $R = 0$

Then, for each table entry X_i:

Step 2i. Generate $mask_i$ for this entry as described above, which should be non-zero only for the table entry to select.

Then, for each word R_j:

Step 3ij. $R_j = R_j \oplus (X_{i,j} \,\&\, mask_i)$

The result will be that one of the multi-word options from the table was selected, assuming that the mask is non-zero for just one of the table entries.

Step 3ij can be efficiently vectorized. Vector instructions make table lookups much faster on x86 server hardware. The Clang compiler seems to be the only one that produces fast vector operations when vector extensions are used in portable C++ code, though GCC purportedly also supports vector extensions. Fortunately x86 server software can statically link to the Snowshoe library and it can therefore be used by code written for other compilers.

4.4.5 Base Field Math

Snowshoe's base field math is all non-zero integer arithmetic modulo the convenient Mersenne prime $p = 2^{127} - 1$. This field is denoted F_p. Integers in F_p are represented in 127 binary digits (bits). The all-ones representation of p is equivalent to zero. Note that p = 1 (mod 3), p = 2 (mod 5), p = 7 (mod 8). -1 = p - 1 is a non-square in F_p. 3 is also non-square in F_p.

During computation numbers are represented internally as 128-bit words, constructed from several machine words of either 32 or 64 bits in size. Each basic math operation takes as input a partially-reduced input that has the high bit clear, but may have the all ones value. This partially-reduced state is allowed for efficiency and is a well known trick.

128-bit big integer math can be implemented using schoolbook methods and then followed by a fast reduction step to bring the result back into partially-reduced form.

4.4.5.1 Reduction after Addition

Alg1: Assuming that partially reduced inputs a, b are provided in the range [0, p] inclusive, addition (r = a + b) produces an intermediate result r.

To reduce the result r:

Step 1. Test the high bit.

Step 2. Clear the high bit.

Step 3. Add the high bit back in at the lowest position of the 128-bit value.

This is perhaps the simplest modular reduction possible, and is ideally implemented with just three x86

instructions: BTRq, ADCq, ADCq.

Subtracting the modulus $p = 2^{127} - 1$ is the same as turning off the high bit and adding one. Can this cause the high bit to be set again? The answer is no. Proof:

The inputs are assumed to be partially reduced, so the largest input values is p, which is technically outside of F_p and equivalent to 0. In this worst case, p + p - p = p, which is also partially reduced and correct. All other cases may be mechanically verified or shown by inspection.

4.4.5.2 Reduction after Subtraction

Alg2: Assuming that partially reduced inputs a, b are provided in the range [0, p] inclusive, subtraction (r = a - b) produces an intermediate result r.

To reduce the result:

Step 1. Test the high bit.

Step 2. Clear the high bit.

Step 3. Subtract the high bit back in from the lowest position of the 128-bit value.

This is ideally implemented with just three x86 instructions: BTRq, SBBq, SBBq. This algorithm also works for reduction following negation.

4.4.5.2.1 Positive case: a ≥ b

The result does not need reduction in this case. The high bit is clear and is already in partially-reduced form.

4.4.5.2.2 Negative case: a < b

The high bit will be set in this case since the high bit is clear in each of the inputs, and a borrow will happen into the high bit position.

In the negative case, we will want to add the modulus back in. Adding the modulus is the same as clearing the high bit and subtracting 1. But can another borrow occur as a result? Answer: Thankfully no, and the proof:

As "a" gets larger, there will always be at least 1 bit set before the high bit, so the subtraction of 1 for reduction will halt borrowing before hitting the high bit.

"0 - p" is the worst case scenario here. In this scenario, the result will be a high bit and low bit set and all other bits will be cleared. Clearing the resulting high bit and subtracting 1 will not cause another borrow. The result will be 0 as expected.

Another example at the other extreme is "0 - 1". In this example all the resulting bits are set. The high bit is set as expected, so reduction logic will trigger to clear the high bit and subtract 1. The result will be a clear high bit, all bits set to 1 except the low bit. This is equal to "p - 1", which is how to represent "-1" in this finite field as expected.

4.4.5.3 Multiplication

Multiplication in F_p is tightly optimized for speed as it is the most expensive operation in the inner loop of all of the curve operations. The resulting optimized code has been rigorously audited to verify that it will work for all input cases. A proof for each step is provided here. Operations such as squaring and multiplying by small constants are special cases of the full multiplication algorithm presented here. The algorithms for these special cases are omitted for brevity but are similarly treated in the source code.

Using the schoolbook method:

```
B1 B0
```

```
          x A1 A0
        ————————
           00 00 <- low product     = A0*B0
        01 01    <- middle products = A0*B1
        10 10    <- middle products + A1*B0
  + 11 11        <- high product    = A1*B1
  ———————————————
```

The middle products' sum fits in 128 bits so that a reduction step can be skipped.

The high half of the low and middle products can be rolled into the next product up without overflow, so the low parts of each are left behind.

Then the high product is shifted left, the high bit of the low half of the 256-bit product is stolen into the high part, and then the high half of the 256-bit product is subtracted from the low half to reduce it to 127 bits.

This reduction approach takes full advantage of the partially reduced inputs to avoid extra operations and appears to be minimal.

Alg3: Multiplication of two partially-reduced inputs A, B in the range [0, p]

Represent A, B as pairs of 64-bit words A_0 (low), A_1 (high), B_0 (low), B_1 (high).

Let R, L, M, and H be 128-bit values that can also be represented as word pairs.

Step 1. Calculate $L = A_0 \cdot B_0$, $M = A_0 \cdot B_1 + A_1 \cdot B_0$, $H = A_1 \cdot B_1$

It is clear that L does not overflow. To show that M does not overflow:

$$M \le 2(2^{64}-1)(2^{63}-1)$$
$$= (2^{64}-1)(2^{64}-2)$$
$$= 2^{128} - 3 \cdot 2^{64} + 2$$
$$\rightarrow M \le 2^{128} - 1$$

To show that H is fully reduced (not just partially reduced):

$$H \le (2^{63}-1)(2^{63}-1)$$
$$= 2^{126} - 2 \cdot 2^{63} + 1$$
$$= 2^{126} - 2^{64} + 1$$
$$\rightarrow H < 2^{127} - 1$$

Incorporate high half of low product into the middle product, so that it can be neglected:

Step 2. $M = M + L_0$

To show that M does not overflow:

$$M + L_0 \le (2^{128} - 3 \cdot 2^{64} + 2) + (2^{64} - 1)$$
$$= 2^{128} - 2 \cdot 2^{64} + 1$$
$$\rightarrow M + L_0 \le 2^{128} - 1$$

Incorporate the high half of middle product into the high product, so that it can be neglected:

Step 3. $H = H + M_1$

At this step, $M_1 \le 2^{64} - 2$ which can be seen by logically right-shifting the upper bound for $M + L_0$ to the right by 64 bits. To show that H is still fully reduced (not just partially reduced):

$$H + M_1 \le (2^{63} - 1)(2^{63} - 1) + (2^{64} - 2)$$
$$= 2^{126} - 2^{64} + 1 + 2^{64} - 2$$
$$\rightarrow H + M_1 \le 2^{126} - 1$$

After step 3 the problem has been reduced to the following and it is clear that the full 256-bit intermediate product has been calculated:

```
          B1 B0
        x A1 A0
        ———————

             00 <- low part of low product remains
          01    <- low part of middle product remains
+ 11 11         <- high product accumulated the rest
————————————————
```

To reduce the 256-bit product, we need to multiply the binary number represented by all of the overflow bits starting from 127 by $p = 2^{127} - 1$, and then subtract the result from the low 128 bits of the product.

This multiplication can be implemented more efficiently through bit shifts. The high product can be doubled, which still fits within 128 bits. The high bit of the low 128 bits can be added to it to create the number represented by the overflow bits.

Incorporate high bit of middle into shifted high bits:

Step 4. $H = (H \ll 1) \,|\, (M_0 \gg 63)$

To show that the result of step 4 is at least partially reduced:

$$H \le 2(2^{126} - 1) + (1)$$
$$= 2^{127} - 2 + 1$$
$$\rightarrow H \le 2^{127} - 1$$

Store low 127 bits in result, as high bit was incorporated above:

Step 5. $R = L_0 + ((M_0 \,\&\, (2^{127} - 1)) \ll 64)$

The result is clearly partially reduced because the high bit was cleared.

Finally, calculated $R = R - p \cdot H$. Observe that this is the same as adding H to R as both are partially reduced.

Step 6. $R = R + H$ (with reduction)

The result is the partially reduced modular product of A,B.

4.4.5.4 Constant-time Operations

The basic addition, subtraction and multiplication algorithms presented above run in constant-time as they compile to series of MUL, ADC, SBB instructions and innocuously unconditional moves.

F_p squaring takes three 64-bit word multiplies instead of 4 since the cross terms are identical but is otherwise the same algorithm as multiplication.

F_p division by 2 is implemented as a multiplication by a constant that is the inverse of 2 in the field.

All of the other expensive operations are implemented using exponential forms rather than variable-time (but faster) GCD evaluation. In practice these expensive operations take less than half the time compared with the base fields for other elliptic curves because these operations are done over fields that are half as large (just 127 bits).

`inv(x)` in F_p runs in constant time using Euler's totient function:

$$\frac{1}{x} = x^{p-2} = x^{2^{127}-3}$$

Using a short addition chain may provide a slight speed improvement using potentially 136 operations. Instead, a simple naive implementation was chosen. Since the field is only 127 bits, inversion takes simply 126 F_p squarings and 12 F_p multiplications evaluated in straight-line sequence in constant time.

`sqrt(x)` in F_p is implemented in constant time using the well known relation:

$\sqrt{x} = x^{\frac{p+1}{4}} = x^{2^{125}}$ Note that square root simply requires 125 squarings in F_p.

The quadratic character function $\chi(x)$ or `chi(x)` in F_p returns:

- **-1** if x does not have a square root in F_p.
- **0** if x is zero.
- **1** if x has a square root.

It is well known that `chi(x)` can be written:

$\chi(x) = x^{\frac{p-1}{2}} = x^{2^{126}-1}$.

And evaluating the exponential is again constant-time. The result will be 0, 1 or p-1, which is adapted to 32-bit integer -1 in the software.

4.4.6 Optimal Extension Field Math

Snowshoe uses grade school complex math denoted F_p^2 to represent the X,Y coordinates of points on the curve. F_p^2 is also known as an optimal extension field of the base field F_p, and in this case the value of $i = \sqrt{-1}$ as in familiar complex arithmetic.

4.4.6.1 Addition

Due to the simplicity of the extension field used in Snowshoe, F_p^2 addition is defined:

$$x + y = (a + ib) + (c + id) = (a + c) + i(b + d)$$

So each addition requires two F_p additions. This also makes addition comparatively slow because two reduction are needed after each F_p^2 addition instead of one for a field that is twice as large. This is mostly offset by the three-opcode reduction step for F_p.

4.4.6.2 Subtraction

F_p^2 subtraction is defined:

$$x - y = (a + ib) - (c + id) = (a - c) + i(b - d)$$

So each subtraction requires two F_p subtractions.

F_p^2 negation is defined:

$$-x = -(a + ib) = -a + i(-b)$$

F_p^2 conjugation is defined:

$$conj(x) = \bar{x} = a - ib$$

4.4.6.3 Multiplication

F_p^2 multiplication is defined:

$$(a + ib) \bullet (c + id)$$
$$= (ac - bd) + i(bc + ad)$$
$$= (ac - bd) + i([a + b][c + d] - ac - bd)$$

Multiplication needs to be especially fast, so its operations were organized to take advantage of deep pipeline architectures like x86.

Alg4: Complex multiplication

Given four F_p temporary variables t_0, t_1, t_2, t_3, and two complex inputs (a,b) and (c,d):

$$t_0 = a + b \quad t_1 = c + d$$

$$t_2 = a \cdot c \quad t_1 = t_0 \cdot t_1 \quad t_3 = b \cdot d$$

$$t_1 = t_1 - t_2$$

$$\mathrm{Re}(r) = t_2 - t_3 \quad \mathrm{Im}(r) = t_1 - t_3$$

This multiplication is similar to the efficiency of Karatsuba applied to a 256-bit field and does not offer any impressive benefits aside from the fast reduction of the base field.

4.4.6.4 Squaring

F_p^2 squaring is defined:

$$(a+ib)\bullet(a+ib)$$
$$=(a^2-b^2)+i(a\cdot b+a\cdot b)$$
$$=(a+b)(a-b)+ib(a+a)$$

Squaring operations are also optimized for ILP. Given four F_p temporary variables t_0, t_1, and t_2:

Alg5: Complex squaring

Given three F_p temporary variables t_0, t_1, t_2, and one complex input (a,b):

$$t_0 = a+b \quad t_1 = a-b \quad t_2 = a+a$$

$$\mathrm{Re}(r) = t_0 \cdot t_1 \quad \mathrm{Im}(r) = t_2 \cdot b$$

Squaring is a less frequent operation, as it is used only for EC point doubling. F_p^2 squarings are unusually efficient, taking only 0.67x the time of an F_p^2 multiply. Usually 256-bit Karatsuba squaring on a more familiar prime field takes 0.75x the time of a multiplication.

4.4.6.5 Inverse

F_p^2 `inv(x)` is defined given x = a + ib:

$$\frac{1}{x} = \frac{\bar{x}}{x\bar{x}} = (a-ib)\cdot(x\bar{x})^{-1} = (a-ib)\cdot(a^2+b^2)^{-1}$$

Alg6: Complex inverse

Given three F_p temporary variables t_0, t_1, t_2, and one complex input (a,b):

$$t_0 = a^2 \quad t_1 = b^2 \quad t_2 = t_0 + t_1$$

$$t_0 = \frac{1}{t_2} \quad t_1 = -b$$

$$\mathrm{Re}(r) = t_0 \cdot a \quad \mathrm{Im}(r) = t_1 \cdot t_0$$

Note that the inverse operation is performed over F_p. As a result `inv(x)` in F_p^2 takes less than half the time as a 256-bit field and affects the overall runtime much less than normal. This inverse is used once to bring EC point coordinates back to affine form for transmission over the network and is often cited as a bottleneck in other elliptic curve implementations.

4.4.6.6 Quadratic Character Function

The quadratic character function $\chi(x)$ or `chi(x)` in F_p^2 is somewhat less easy to understand since there is no obvious definition of positive or negative in a 2 dimensional space so there is no "principal square root."

It is well known that `chi(x)` can be written using the identity $x\bar{x} = x^1 x^p = x^{1+p}$:

$$\chi(x) = x^{\frac{p^2-1}{2}} = x^{\frac{p-1}{2}(1+p)} = \left(x\overline{x}\right)^{\frac{p-1}{2}} = \chi(x\overline{x}) = \chi(a^2+b^2).$$

This reduces the evaluation of `chi(x)` to the base field.

This function is only used once during Elligator point decoding, which is the full domain hash function that enables Tabby PAKE.

4.4.6.7 Square Root

Square roots in even extension fields are unusually expensive, even in the familiar grade school complex math used in Snowshoe. The best algorithm that could be found is called the "Complex method"[38] and is reproduced here.

$$d_0 = d_1 = \frac{a+|x|}{2} \quad d_{-1} = \frac{a-|x|}{2}$$

$$\mathrm{Re}(r) = \sqrt{d_{\chi(d_1)}} \quad \mathrm{Im}(r) = \frac{b}{2\sqrt{d_{\chi(d_1)}}}$$

Alg6: Complex method of square root

Given temporary variables c, d, e, t and input complex value (a, b):

Step 1. Calculate c= |x|:

$$c = a^2 \quad t = b^2 \quad c = c + t \quad c = \sqrt{c}$$

Step 2. Calculate d, e:

$$d = a + c \quad d = \frac{d}{2}$$

$$e = a - c \quad e = \frac{e}{2}$$

Step 3. Calculate `chi(d)`. This can be done with the method above.

Step 4. Select between d, e based on `chi(d)` in constant time. If the result was negative, e replaces d. Otherwise d continues unchanged after this step.

The fastest portable way to implement this constant-time conditional move seems to be to create a 64-bit mask by sign-extending the 32-bit result of chi() up to 64 bits. The result is either all ones or all zeroes in the mask word(s) and then:

$$d_0 = d_0 \oplus (d_0 \oplus e_0)\,\&\, mask$$
$$d_1 = d_1 \oplus (d_1 \oplus e_1)\,\&\, mask$$

This sort of conditional move is used throughout the Snowshoe code where needed because it is both portable and vectorizes well. However, even though it executes in constant time this sort of conditional move is not safe from high-resolution power analysis as the mask setup is visible on a power trace.

Step 5. Calculate Re(r) and Im(r):

$$\mathrm{Re}(r) = \sqrt{d}$$

$$t = \mathrm{Re}(r) + \mathrm{Re}(r) \quad t = \frac{1}{t} \quad \mathrm{Im}(r) = b \cdot t$$

The result is one of the square roots of the input. Note that there are always two solutions to a square root, and the complex negation of this result provides a second result. However, it seems to be impossible to tell which of the two solutions is provided by this algorithm.

This function is only used once during Elligator point decoding. The ambiguity of this solution to the square root is addressed as part of that algorithm.

4.4.7 Elligator Point Decoding

The Elligator point decoding function is implemented based on the original article[39] by Bernstein et al. This function deterministically decodes each (random in Tabby) 255-bit input into a unique point on the Snowshoe curve, for which the generator scalar (private key) is unknown. As far as the author is aware, Snowshoe is the first public domain software that attempts to implement any part of Elligator for a real application.

While it is guaranteed to be on the curve, the decoded point may not be a point of order q. As with normal point multiplication, the point can be simply multiplied by 4 to transform it into a uniformly distributed point of order q. This is discussed further in the Tabby PAKE specification in a subsequent section of this document.

The output point can also be invalid as input for Snowshoe functions. For instance, Snowshoe EC point multiplication rejects input with x = 0, which can happen if the input to the Elligator point decoding function is zero for example. This is discussed further in the Tabby PAKE specification, but it is easy to try again by generating a new random input in this extremely rare case. Due to the design of the Cymric generator this will not reveal any information about the next number that gets generated, so there is no dangerous timing side channel. This is so unlikely that it may never happen.

The Snowshoe curve uses F_p^2 math and Elligator point decoding involves the ambiguous F_p^2 square root function discussed previously. A simple modification to the original Elligator algorithm was made to allow it to allow it to be treated as a full domain hash function. The modified Elligator point decoding function is presented here.

Alg7: Snowshoe Elligator Point Decoding algorithm

Given a 256-bit input value W represented as four 64-bit words W_0, W_1, W_2, W_3, from low to high bit positions.

The low bit of W_1 is discarded. The low bit of W_3 is referred to as the SIGNBIT.

The input 256-bit value W is then transformed into a complex number U in F_p^2:

Step 1. $\mathrm{Re}(U) = W_0 \mid ((W_1 \gg 1) \ll 64)$

$\mathrm{Im}(U) = W_2 \mid ((W_3 \gg 1) \ll 64)$

Essentially this is keeping the low word, but clipping the first bit out of the high word of each pair. This keeps 127 uniformly distributed bits for each of the two F_p components of A.

Because the base field F_p is the 127-bit integers modulo $p = 2^{127} - 1$, there is a rare case where either Re(A) or Im(A) will be equal to p, which is the same as 0 in F_p. This rare case causes a slight acceptable bias towards (0,0) in the output if the input is uniformly random. Otherwise the input is uniformly distributed, and since p = 1 mod 4, all input values in the field will map to curve points according to the Elligator article.

Recall the Snowshoe curve equation: $E : a \cdot u \cdot x^2 + y^2 = 1 + d \cdot u \cdot x^2 \cdot y^2$

The Elligator curve equation is in Weierstrass coordinates: $t^2 = s^3 + A \cdot s^2 + B \cdot s$, where $A = -(a+d)u$ and $B = a \cdot d \cdot u^2$.

The s-coordinate is temporarily chosen based on input scalar U.

Step 2. $z = -\dfrac{A}{1 + u \cdot U^2}$

Solving for t given s involves taking a square root, so the chi function must be checked to determine which of the two square root solutions is selected:

Step 3. $e = \chi(z^3 + A \cdot z^2 + B \cdot z)$

Then e and z are used to select the s-coordinate on the Weierstrass curve:

Step 4. $s = e \cdot z - \dfrac{(1-e)A}{2}$

The value of t^2 is calculated from s using the Elligator Weierstrass curve equation:

Step 5. $t^2 = s^3 + A \cdot s^2 + B \cdot s$

The final Snowshoe curve Y coordinate can then be calculated:

Step 6. $P_Y = \dfrac{t^2 - (a-d) \cdot u \cdot s^2}{t^2 + (a-d) \cdot u \cdot s^2}$

For each Y coordinate there are two possible X coordinates. The Snowshoe curve equation is used to solve for the X coordinate:

Step 7. $P_X = \sqrt{\dfrac{{P_Y}^2 - 1}{d \cdot u \cdot {P_Y}^2 - a \cdot u}}$

Note that P_Y is squared while solving for P_X. This eliminates half the possible P_X values and would prevent this from being a true full domain hash function.

The F_p^2 "complex method" square root algorithm used by Snowshoe also does not have a natural signedness unlike familiar integer square roots, so it does not seem to be possible to determine which of the two results it is returning.

The only guarantee is that the negation of P_X is always also a second valid point that would otherwise not be in the range of the Elligator function. It is clear from the symmetrical visual shape of a twisted Edwards curve and the curve equation that $+P_X$, $-P_X$ and $+P_Y$, $-P_Y$ are all valid points.

Therefore, to make this a full domain hash function, the sign of the X coordinate is flipped based on the value of the SIGNBIT. When the input is uniform random, this ensures that all points on the curve are equally likely apart from the previously noted exception at 0.

Step 8. P_X is conditionally negated when SIGNBIT is nonzero.

To implement this in constant time, a 64-bit mask is formed that is all ones when SIGNBIT is nonzero, and all zeroes when SIGNBIT is zero. The negation of P_X is calculated in a temporary buffer and conditionally assigned to P_X:

$$X_0 = X_0 \oplus (X_0 \oplus (-X)_0) \,\&\, mask$$
$$X_1 = X_1 \oplus (X_1 \oplus (-X)_1) \,\&\, mask$$

The resulting decoded point P corresponds uniquely to the input value W.

The Elligator point encoding function is not implemented in Snowshoe, partly because it is not deterministic but mainly because it is not needed for Tabby.

4.4.8 Elliptic Curve Point Doubling

The extended Twisted Edwards coordinates are used for curve operations.

EC point doubling in these coordinates is written as:

$(\mathrm{X}_2, \mathrm{Y}_2, \mathrm{T}_{2a}, T_{2b}, \mathrm{Z}_2) = 2 * (\mathrm{X}, \mathrm{Y}, \mathrm{Z})$, where $T_{2a} \cdot T_{2b} = \frac{X_2 Y_2}{Z_2}$.

This doubling formula is based on the dedicated doubling formula from the article[40] that introduced this coordinate system. The algorithm is optimized for Instruction-Level Parallelism (ILP) and low register usage.

The original formula appears to produce all negative results, which is okay. It was adjusted to produce the same as the math expressions in the paper, which works out better for my formulae.

There are alternative formulae introduced with the Ted1271gls curve by Longa that do not seem preferable for my base field. The EFD website also has a version assuming Z = 1 that is more efficient, but it would add the complexity of a whole new EC double function, which would run counter to the simplicity goal.

This doubling formula produces a split value for T similar to Hamburg's fast and compact implementation. The value of T can be reconstructed after EC point doubling by multiplying the two split halves.

This lazier calculation is preferable to generating T in this function, since the value of T is not always needed afterwards. For instance if doubling is the final operation, or if doubling is followed by another doubling. This also makes it much easier to implement EC point multiplication in constant time.

Alg8: Extended twisted Edwards point doubling

Given input point (X, Y, Z), temporary variable W, and output point $(\mathrm{X}_2, \mathrm{Y}_2, \mathrm{T}_{2a}, T_{2b}, \mathrm{Z}_2)$:

Step 1. $T_{2a} = X + Y$

Step 2. $Z_2 = Z^2$

Step 3. $X_2 = X^2$

Step 4. $T_{2a} = T_{2a}{}^2$

Step 5. $Y_2 = Y^2$

Step 6. $T_{2a} = T_{2a} - X_2$

After this step, $T_{2a} = (X+Y)^2 - X^2 = Y^2 + 2 \cdot X \cdot Y$.

Step 7. $X_2 = u \cdot X_2$

In step 7, $u = (2 + i)$ is a curve parameter. After this step, $X_2 = u \cdot X^2$.

Step 8. $Z_2 = Z_2 + Z_2$

After this step, $Z_2 = 2 \cdot Z^2$.

Step 9. $W = Y_2 - X_2$

After this step, $W = Y^2 - u \cdot X^2$.

Step 10. $T_{2a} = T_{2a} - Y_2$

After this step, $T_{2a} = 2 \cdot X \cdot Y$.

Step 11. $Z_2 = Z_2 - W$

After this step, $Z_2 = 2 \cdot Z^2 - Z^2 + u \cdot X^2$.

Step 12. $T_{2b} = Y_2 + X_2$

After this step, $T_{2b} = Y^2 + u \cdot X^2$.

Step 13. $X_2 = T_2 \cdot Z_2$

After this step, $X_2 = 2 \cdot X \cdot Y \cdot (2 \cdot Z^2 - Y^2 + u \cdot X^2)$.

Step 14. $Y_2 = W \cdot T_{2b}$

After this step, $Y_2 = (Y^2 - u \cdot X^2) \cdot (Y^2 + u \cdot X^2)$.

Step 15. $Z_2 = W \cdot Z$

After this step, $Z_2 = (Y^2 - u \cdot X^2) \cdot (2 \cdot Z^2 - Y^2 + u \cdot X^2)$.

And the result is returned in $(\mathrm{X}_2, \mathrm{Y}_2, \mathrm{T}_{2a}, T_{2b}, \mathrm{Z}_2)$.

The value of $T_2 = T_{2a} \cdot T_{2b} = 2 \cdot X \cdot Y \cdot (Y^2 + u \cdot X^2)$ but this it not performed here and it is preferred to leave it in an unreduced state for speed in case it is not needed.

In terms of F_p^2 operations this algorithm uses 4 squarings, 3 multiplications, 7 additions, and 1 multiplication by the curve constant u. If Z=1 then it saves 1 squaring and 1 addition.

4.4.9 Elliptic Curve Point Addition

The extended Twisted Edwards coordinates are used for curve operations.

EC point addition in these coordinates is written as:

$$P_3 = (X_3, Y_3, T_{3a}, T_{3b}, Z_3) = (X_1, Y_1, T_{1a}, T_{1b}, Z_1) + (X_2, Y_2, T_2, Z_2) = P_1 + P_2$$

The split values are related so that $T_1 = T_{1a} \cdot T_{1b}$. Note that the second point has a precomputed value for T_2. This is arranged to always be the case, and T_2 is usually taken from a precomputed table during EC point multiplication.

The Snowshoe EC point addition algorithm is **unified**, meaning that if the two input points are equivalent, then the result is still correct. The point addition algorithm is also **complete**, meaning that any point on the curve may be used as input.

Alg9: Extended twisted Edwards point addition

Given input point $P_1 = (X_1, Y_1, T_{1a}, T_{1b}, Z_1)$ and input point $P_2 = (X_2, Y_2, T_2, Z_2)$, temporary variables W_1, W_2, and output point $P_3 = (X_3, Y_3, T_{3a}, T_{3b}, Z_3)$:

Step 1. $T_{3a} = T_{1a} \cdot T_{1b}$

Step 2. $T_{3b} = X_1 \cdot Y_1$

Step 3. $W_1 = X_2 + Y_2$

Step 4. $W_2 = T_{3a} \cdot T_2$

Step 5. $T_{2b} = T_{2b} \cdot W_1$

After step 5, $T_{2b} = (X_1 + Y_1) \cdot (X_2 + Y_2)$.

Step 6. $T_{3a} = X_1 \cdot X_2$

Step 7. $Y_3 = Y_1 \cdot Y_2$

Step 8. $W_2 = u \cdot W_2$

After step 8, $W_2 = u \cdot T_1 \cdot T_2$.

Step 9. $T_{3b} = T_{3b} - T_{3a}$

After step 9, $T_{3b} = (X_1 + Y_1) \cdot (X_2 + Y_2) - X_1 \cdot X_2$.

Step 10. $W_2 = d \cdot W_2$

After step 10, $W_2 = u \cdot d \cdot T_1 \cdot T_2$.

Step 11. $T_{3a} = u \cdot T_{3a}$

After step 11, $T_{3a} = u \cdot X_1 \cdot X_2$.

Step 12. $T_{3b} = T_{3b} - Y_3$

After step 12, $T_{3b} = (X_1 + Y_1) \cdot (X_2 + Y_2) - X_1 \cdot X_2 - Y_1 \cdot Y_2 = (X_1 \cdot Y_2 + Y_1 \cdot X_2)$.

Step 13. $Z_3 = Z_1 \cdot Z_2$

Note that step 13 can be skipped if $Z_2 = 1$.

Step 14. $T_{3a} = T_{3a} + Y_3$

After step 14, $T_{3a} = Y_1 \cdot Y_2 + u \cdot X_1 \cdot X_2$.

Step 15. $W_1 = Z_3 - W_2$

After step 15, $W_1 = Z_1 \cdot Z_2 - d \cdot u \cdot T_1 \cdot T_2$.

Step 16. $Z_3 = Z_3 + W_2$

After step 16, $Z_3 = Z_1 \cdot Z_2 + d \cdot u \cdot T_1 \cdot T_2$.

Step 17. $X_3 = T_{3b} \cdot W_1$

After step 17, $X_3 = (X_1 \cdot Y_2 + Y_1 \cdot X_2) \cdot (Z_1 \cdot Z_2 - d \cdot u \cdot T_1 \cdot T_2)$.

Step 18. $Y_3 = Z_3 \cdot T_{3a}$

After step 18, $Y_3 = (Z_1 \cdot Z_2 + d \cdot u \cdot T_1 \cdot T_2) \cdot (Y_1 \cdot Y_2 + u \cdot X_1 \cdot X_2)$.

Note that $T_3 = T_{3a} \cdot T_{3b} = (X_1 \cdot Y_2 + Y_1 \cdot X_2) \cdot (Y_1 \cdot Y_2 + u \cdot X_1 \cdot X_2)$, though this is not computed as part of the formula.

Step 19. $Z_3 = Z_3 \cdot W_1$

After step 19, $Z_3 = (Z_1 \cdot Z_2 - d \cdot u \cdot T_1 \cdot T_2) \cdot (Z_1 \cdot Z_2 + d \cdot u \cdot T_1 \cdot T_2)$.

And the result is returned in $(X_3, Y_3, T_{3a}, T_{3b}, Z_3)$.

For Twisted Edwards elliptic curves there are faster addition laws when $a \cdot u = -1$ and when the algorithm does not need to be unified. However, for the Snowshoe curve, $a \cdot u = -2 - i$ and the formulas cannot be rearranged to yield a simpler formula. Also, unified formulas are required for EC point multiplication involving endomorphisms because there is no guarantee that the same point will not be added to itself. Anyhow, the difference in performance would be just one F_p^2 multiplication, which is not incredibly compelling for extra complexity even if it was possible.

In terms of F_p^2 operations this algorithm uses 9 multiplications, 7 additions, 2 multiplications by the curve constant u, and 1 multiplication by the curve constant d. If $Z_2 = 1$ then 1 multiplication is saved.

4.4.10 Elliptic Curve Point Multiplication

4.4.10.1 Generator-Base Multiplication

Generator-base multiplication refers to the evaluation of: R=[4]kG, where G is the generator point, and **0 < k < q**. The multiplication by the cofactor 4 is optional since the generator point is trusted to be a point of order q.

Multiplication by the generator point is implemented using the LSB-set comb method (introduced by Hernandez et al[41] in 2013) with w=6, v=7. These values of w, v were arrived at by exhaustively testing all of the options and selecting the fastest parameters.

The full details of the LSB-set comb method are somewhat outside of the scope of this document. Most importantly, by using the Constant-Time Operations approaches as discussed earlier it executes in constant time. Interestingly, the GLS endomorphism does not help with multiplications by the generator point but also does not lead to any performance penalty.

Generator-base multiplication requires 42 EC point additions, 5 EC point doubles, and 6 large point table lookups.

There are other methods such as the MSB-set comb preferred by Hamburg that are roughly similar in speed and may be slightly faster.

4.4.10.2 Variable-Base Multiplication

Generator-base multiplication refers to the evaluation of: R=4kP, where P is a point on the curve, and **0 < k < q**.

This is implemented using the GLV-SAC method (also introduced by Hernandez et al in 2013) with m=2. This selection of m is due to the single GLS endomorphism. This is the most efficient constant-time approach for table-based EC point multiplication.

Rather than going into the full details of this multiplication algorithm, just an overview is provided here:

Given P, and k, compute R = 4kP:

Step 1. Decompose scalar k into sub-scalars k1, k2.

Step 2. Q = endo(P)

Step 3. Conditionally negate P based on the sign of k1.

Step 4. Conditionally negate Q based on the sign of k2.

Step 5. Expand the points into extended coordinates.

Step 6. Precompute an 8 point GLV-SAC table from P, Q for multiplication.

Step 7. Recode sub-scalars k1, k2 for GLV-SAC.

Step 8. Set working point R to the table entry selected by bits 126, 127 of the recoded sub-scalars.

For each set of two bits numbered 125,124 down to 1,0:

Step 9i. Select a point T from the table as indicated by the sub-scalars k1, k2.

Step 10i. R = 4R

Step 11i. R = R + T

Finally:

Step 12. Conditionally set R = R + P if required by GLV-SAC.

Step 13. R = 4R

Step 13 ensures that the result is a point of order q even if the input was not.

Step 14. Convert R to 64 byte affine (X,Y) form.

Converting to affine form is extremely efficient on the Snowshoe curve, requiring fewer than half the usual number of operations, due to the efficiency of F_p^2 inversion. With most curves this is a considerable part of the computation.

Variable-base multiplication requires overall 70 EC point adds, 129 EC point doubles, and 64 small point table lookups.

4.4.10.3 Mixed-Base Multiplication

Mixed-base multiplication refers to the evaluation of: R = 4aP + 4bG, where G is the generator point, and **0 < a,b < q**. The multiplication by the cofactor 4 is required since the variable point P is not trusted to be a point of order q.

Interleaving the EC point additions from Generator-base multiplication with those from Variable-base multiplication is a straight-forward approach that works.

The tuning (v,w) for the LSB-set comb method is different in this case. The table should stay at 128 points since that is an optimal memory access trade-off. But there is no need to use multiple tables (v = 1) since the EC point double operations are "free" since they need to be performed regardless for the Variable Base operations that are interleaved. As a result w = 8, and since 8 does not evenly divide 252 it is not necessary to do the final correction addition step, which simplifies the algorithm. For this tuning, there are just 32 EC point adds required for the generator multiplication.

Since all the point operations are linear, it is possible to interleave the Generator-base and Variable-base multiplication operations even though the overall number of doubles for each differs. Introducing extra adds for the Generator-base multiplication near the end of the evaluation loop for the Variable-base multiplication still exhibits a regular pattern. The final conditional negation from the Variable-base multiplication is merged into the rest of the EC adds by conditionally inverting the sign of each added point instead to avoid making the interleaving more complex.

The endomorphism is used as before to speed up the multiplication. Overall the cost is about the same as one m=2 GLV-SAC multiplication with just 32 extra EC adds from large point table lookups.

4.4.10.4 Double-Base Multiplication

Double-base multiplication refers to the evaluation of: R = 4aP + 4bQ, where P,Q are points on the curve, and **0 < a,b < q**. The multiplication by the cofactor 4 is required since the variable points P,Q are not trusted to be points of order q.

The GLV-SAC approach is used as in Variable-base multiplication, excepting that m=4.

Rather than going into the full details of this multiplication algorithm, just an overview is provided here:

Given P,Q, and a,b, compute R = 4aP + 4bQ:

Step 1. Decompose scalars a,b into sub-scalars a0, a1, b0, b1.

Step 2. P' = endo(P), Q' = endo(Q)

Step 3. Conditionally negate P, P', Q, Q' based on the sign of a0, a1, b0, b1.

Step 4. Expand the points into extended coordinates.

Step 5. Precompute an 8 point GLV-SAC m=4 table for multiplication.

Step 6. Recode sub-scalars a0, a1, b0, b1 for GLV-SAC.

Step 7. Set working point R to the table entry selected by bits at offset 126.

For each bit from 125 down to 0:

Step 8i. Select a point T from the table as indicated by the 4 sub-scalars.

Step 9i. R = 2R

Step 10i. R = R + T

Finally:

Step 11. Conditionally set R = R + P if required by GLV-SAC.

Step 12. R = 4R

Step 12 ensures that the result is a point of order q even if the input was not.

Step 13. Convert R to 64 byte affine (X,Y) form.

Double-base multiplication requires overall 134 EC point adds, 128 EC point doubles, and 127 small point table lookups.

Implementing this using the sum of two Variable-base multiplications would cost twice as many doubles and a few more adds and other small operations, making this version worthwhile for performance.

4.4.11 Snowshoe Review

Unfortunately no existing software is available that implements the elliptic curve operations such as Elligator or EC double-base point multiplication required for the Tabby protocol, and so it regrettably had to be written.

As a side-effect of implementing the necessary elliptic curve operations, the performance of Tabby is unusually excellent thanks to the use of the Snowshoe curve's built-in GLS endomorphism. It is not the fastest curve possible, but Snowshoe is the fastest general-purpose public domain software for elliptic curves at this time. To some, this may make the extra effort worthwhile.

Armed with the powerful Snowshoe and Cymric libraries, the Tabby protocol has a full set of fast and constant-time operations to apply to the PAKE problem.

5 Conclusions

This new approach to PAKE provided by the Tabby software enables new high-speed applications by bringing the performance of mutual password-based authentication back up to par with other high speed cryptography.

Tabby PAKE provides strong security guarantees that are not often seen in existing applications. Since nearly all web apps are based on username/password logins, mutual password-based authentication should naturally be built into browsers. This can be a large part of the solution to the erosion of trust in HTTPS for protecting secure areas of websites. Previously the low throughput of strong cryptography may have hindered its adoption for this task.

Software code is available at http://tabbypake.com

---

[1] B-SPEKE articles (Jablon 1997) http://www.jablon.org/jab96.pdf,

http://www.jablon.org/jab97.pdf

[2] SPEKE patents: https://www.google.com/patents/US6226383,

https://www.google.com/patents/US6792533

[3] Official SRP design website http://srp.stanford.edu/design.html

[4] SPAKE2 article (Abdalla Poincheval 2005) http://www.di.ens.fr/~abdalla/papers/AbPo05a-letter.pdf

[5] AugPAKE article (Shin 2008) http://eprint.iacr.org/2010/334.pdf

[6] IETF proposal for integration into TLS (Shin) https://datatracker.ietf.org/doc/draft-shin-tls-augpake/

[7] AugPAKE patent (Shin) https://datatracker.ietf.org/ipr/1284/

[8] J-PAKE article (Hao Ryan 2010) http://eprint.iacr.org/2010/190.pdf

[9] "Elligator: Elliptic-curve points indistinguishable from uniform random strings" (Bernstein Hamburg Krasnova Lange 2013) http://elligator.cr.yp.to/elligator-20130828.pdf

[10] Cfrg post on applying Elligator to SPAKE2 (Hamburg 2014) http://www.ietf.org/mail-archive/web/cfrg/current/msg03840.html

[11] "Lyra: Password-Based Key Derivation with Tunable Memory and Processing Costs" (Almeida et al 2014) http://eprint.iacr.org/2014/030.pdf

[12] OpenWall newsgroup post about Lyra parameters (Solar Designer 2014) http://www.openwall.com/lists/crypt-dev/2014/01/13/1

[13] "Entropy Attacks!" (Bernstein 2014) http://blog.cr.yp.to/20140205-entropy.html

[14] "Simple Password-Based Encrypted Key Exchange Protocols" (Abdalla, Pointcheval 2005) http://www.di.ens.fr/~abdalla/papers/AbPo05a-letter.pdf

[15] DragonSRP (Slechta 2012) https://github.com/slechta/DragonSRP

[16] ECRYPT II Key Length Recommendations http://www.keylength.com/en/3/

[17] Github: Cymric http://github.com/catid/cymric

[18] Github: Snowshoe http://github.com/catid/snowshoe

[19] "BLAKE2: simpler, smaller, fast as MD5" (Aumasson, Neves, Zooko, CodesInChaos 2013) http://blake2.net/blake2_20130129.pdf

[20] "Lyra: Password-Based Key Derivation with Tunable Memory and Processing Costs" (Almeida Andrade Barreto Simplicio 2014) http://eprint.iacr.org/2014/030.pdf

[21] The ChaCha Cipher Family (Bernstein) http://cr.yp.to/chacha.html

[22] "Safe Curves" (Bernstein et al 2013) http://safecurves.cr.yp.to/

[23] "Elliptic and Hyperelliptic Curves: a Practical Security Analysis" (Bos Costello Miele 2013) http://eprint.iacr.org/2013/644.pdf

[24] "Keep Calm and Stay with One" (Hernandez Longa Sanchez 2013) http://eprint.iacr.org/2013/158

[25] "Endomorphisms for Faster Elliptic Curve Cryptography on a Large Class of Curves" (Galbraith Lin Scott 2008) http://eprint.iacr.org/2008/194

Updated: Endomorphisms for Faster Elliptic Curve Cryptography on a Large Class of Curves" (Galbraith Lin Scott 2009) http://www.iacr.org/archive/eurocrypt2009/54790519/54790519.pdf

[26] "Families of fast elliptic curves from Q-curves" (Smith 2013) http://eprint.iacr.org/2013/312.pdf

[27] "Faster Compact Diffie-Hellman: Endomorphisms on the x-line" (Costello Hisil Smith 2013) http://eprint.iacr.org/2013/692.pdf

[28] "Fast and compact elliptic-curve cryptography" (Hamburg 2012) http://eprint.iacr.org/2012/309.pdf

[29] "EFD: Genus-1 curves over large-characteristic fields" (Lange et al) http://www.hyperelliptic.org/EFD/g1p/index.html

[30] "Keep Calm and Stay with One" (Hernandez Longa Sanchez 2013) http://eprint.iacr.org/2013/158

[31] "Analysis of Efficient Techniques for Fast Elliptic Curve Cryptography on x86-64 based Processors" (Longa Gebotys 2010) http://print.iacr.org/2010/335

[32] MAGMA Online Calculator http://magma.maths.usyd.edu.au/calc/

[33] "Easy scalar decompositions for efficient scalar multiplication on elliptic curves and

genus 2 Jacobians" (Smith 2013) http://hal.inria.fr/docs/00/87/49/25/PDF/easy.pdf

[34] "Division by Invariant Integers using Multiplication" (Granlund Montgomery 1991) http://pdf.aminer.org/000/542/596/division_by_invariant_integers_using_multiplication.pdf

[35] "Curve25519: new Diffie-Hellman speed records" (Bernstein 2006) http://cr.yp.to/ecdh/curve25519-20060209.pdf

[36] "Kummer strikes back: new DH speed records" (Berstein et al 2014) http://cr.yp.to/hecdh/kummer-20140218.pdf

[37] "Keep Calm and Stay with One" (Hernandez Longa Sanchez 2013) http://eprint.iacr.org/2013/158

[38] "Square root computation over even extension fields" (Adj Henriquez 2012) http://eprint.iacr.org/2012/685.pdf

[39] "Elligator: Elliptic-curve points indistinguishable from uniform random strings" (Bernstein Hamburg Krasnova Lange 2013) http://elligator.cr.yp.to/elligator-20130828.pdf

[40] "Twisted Edwards Curves Revisited" (Hisil Wong Carter Dawson 2008) http://www.iacr.org/archive/asiacrypt2008/53500329/53500329.pdf

[41] "Keep Calm and Stay with One" (Hernandez Longa Sanchez 2013) http://eprint.iacr.org/2013/158