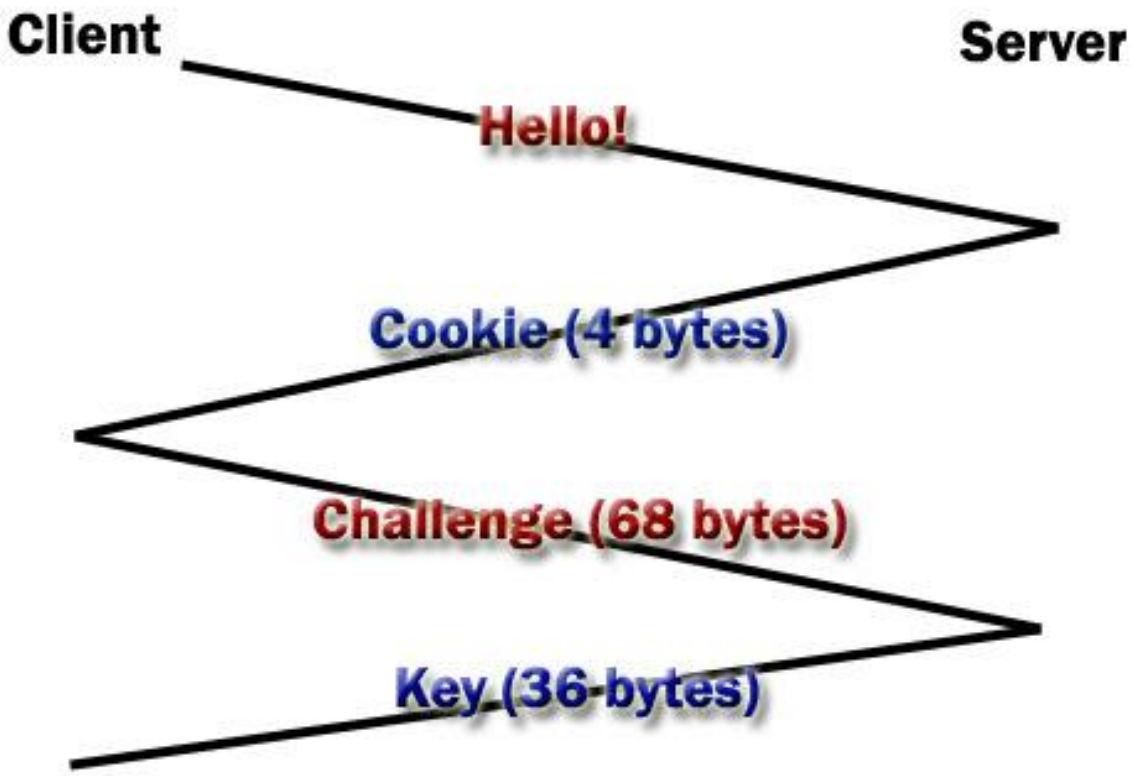
cat::Tunnel

Tunnel puts together the cryptographic primitives provided by libcatcrypt to easily create a secure tunnel over the Internet.

It is a four-way handshake:



After the handshake completes, both sides can communicate securely over the internet.

Interesting specifications:

- DESIGNED FOR CUSTOM UDP/IP PROTOCOLS
- 256-BIT/384-BIT ELLIPTIC CURVE DIFFIE-HELLMAN (ECDH) ASYMMETRIC ENCRYPTION
- 256-BIT/384-BIT SYMMETRIC KEY ENCRYPTION
- 256-BIT SUITABLE FOR SECRET AND 384-BIT SUITABLE FOR TOP SECRET COMMUNICATIONS
- MESSAGE AUTHENTICATION INCLUDED

How to #include and link Tunnel code

1. **Extract the libcat code and build the static libraries for your target.**
2. **#include <cat/AllTunnel.hpp>**
3. **Link with:**
   /lib/cat/libcatcommon-m##-1.0.lib
   /lib/cat/libcatmath-m##-1.0.lib
   /lib/cat/libcatcrypt-m##-1.0.lib
   /lib/cat/libcattunnel-m##-1.0.lib
   (m = d for debug, r for release. ## = 32 for 32-bit, 64 for 64-bit)

The header file <cat/crypt/tunnel/Tunnel.hpp> looks like this (heavily abbreviated):

```
class TunnelSession
{
	bool Decrypt(u8 *buffer, int buf_bytes);
	void Encrypt(u8 *buffer, int msg_bytes);
};

class TunnelServer
{
	void Initialize(MersenneTwister *, const EdPrivateKey *);
	void GenerateCookie(u32 ip, u16 port, u8 *buffer);
	bool ValidateChallenge(u8 *buffer, u32 ip, u16 port, EdSharedSecret *);
	void GenerateKeyResponse(EdSharedSecret *, MersenneTwister *,
				 TunnelSession *, u8 *buffer);
};

class TunnelClient : public TunnelSession
{
	void Initialize(const EdServerPublicKey *ServerPublicKey);
	void GenerateChallenge(MersenneTwister *prng);
	void FillChallenge(u8 *cookie_buffer, u8 *challenge_buffer);
	bool ProcessKeyResponse(u8 *buffer);
};
```

A single TunnelServer object should be instantiated on the server side.
One TunnelSession object should be instantiated for each connected client.

A single TunnelClient object should be instantiated on the client side.

Protocol

OFFLINE

A server has a private key and a public key. They are generated by invoking TwistedEdwardServer::GenerateOfflineStuff().

Example code:

```
MersenneTwister prng;
prng.init();

EdPrivateKey server_private_key;
EdServerPublicKey server_public_key;
TwistedEdwardServer::GenerateOfflineStuff(&prng, &server_private_key,
                                          &server_public_key);
```

Before the server starts up, it needs to load its own private key. It can do this either from disk or compiled into the executable.

Example server-side code:

```
MyTunnelServer.Initialize(prng, server_private_key);
```

The PRNG used by libcatmath is the Mersenne Twister. A PRNG object reference is passed into each of the functions so that the generator usage can be thread-safe. I envision one PRNG object to be allocated for each thread and kept in Thread Local Storage (TLS).

Before the client tries to connect to the server, it must be told the server's public key.

Example server-side code:

```
MyTunnelClient.Initialize(server_public_key);
```

The server MUST NOT send its own public key during the protocol. It should either be compiled into the client executable or transmitted over a secure connection from a trusted server. The server's private key must be kept in a secure location.

C2S HELLO

The HELLO message can contain anything you want. It should contain some unique identifier for your protocol. For example, if HELLO = "Phoenix Protocol", then the server can verify that the HELLO is from a client that is running the correct protocol.

Right after the client sends the HELLO message, the client should start calculating a CHALLENGE message by invoking the TunnelClient. GenerateChallenge() method.

The client MUST generate a new CHALLENGE every time it connects.

Example client-side code:

```
MyTunnelClient.GenerateChallenge(prng);
```

After receiving the HELLO, the server MUST NOT allocate any memory for the new client. Instead, the server will create a COOKIE by invoking the TunnelServer.GenerateCookie() method and send it to the client. This handshake is initially stateless.

Example server-side code:

```
u8 response[TUNNEL_COOKIE_BYTES];
MyTunnelServer.GenerateCookie(source.ip, source.port, response);
Send response, sizeof(response)
```

S2C COOKIE

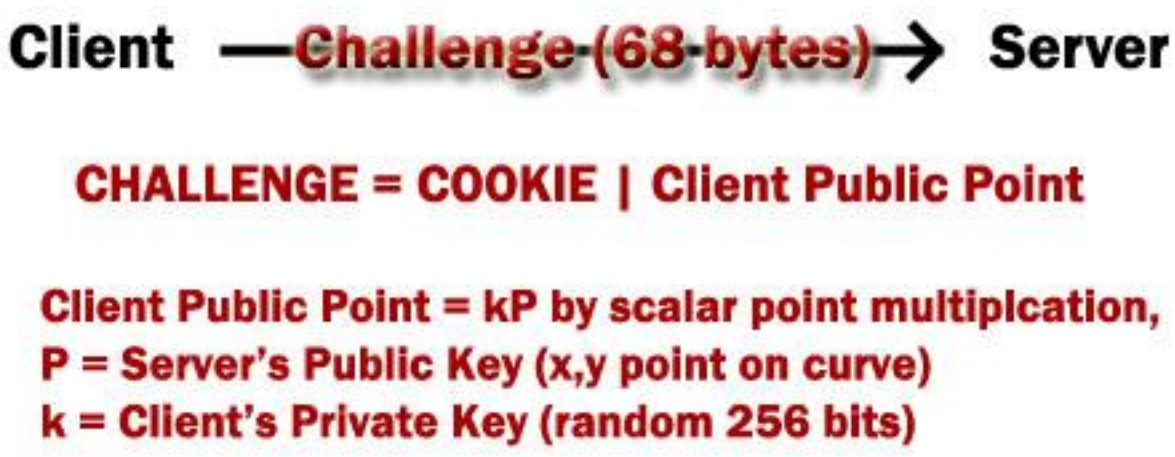
The server's COOKIE is a Salsa6 hash of the connecting client's IP address and port with the current epoch. Salsa6 is keyed with a random 512-bit number on server startup. The epoch changes every 250 milliseconds by default. So every 250 milliseconds each client will be assigned a new unique cookie. The cookie includes a bin number 0..15 that allows the server to remember up to 4 seconds of cookies. After 4 seconds the server will no longer accept a cookie from the client.

When the client receives the server's cookie, it will immediately respond with a CHALLENGE message that was previously computed.

Example client-side code:

```
u8 response[TUNNEL_CHALLENGE_BYTES];
MyTunnelClient.FillChallenge(COOKIE, response);
Send response, sizeof(response)
```

C2S CHALLENGE



Internally the public keys are points on the elliptic curve. For computation, I use Extended Twisted Edwards coordinates (X:Y:T:Z) on a Twisted Edwards curve with a = -1 and d = 321 over Fp with p = 2^256-189. When transmitted or stored, the points are compressed to affine coordinates (x,y). Each coordinate x or y is 256 bits (32 bytes), meaning that a coordinate pair is 64 bytes.

When the server receives the CHALLENGE message, it can determine whether or not the message is valid by invoking the TunnelServer. ValidateChallenge() method.

This method may take up to 1 millisecond on a 32-bit build and 2.5 GHz Intel Xeon processor.

Typically using a 64-bit operating system and 64-bit server build will cut the processing time in half. I recommend compiling libcatmath with full optimization for a 64-bit platform. Compiling with the Intel C++ Compiler provides the best performance. The client-side of the protocol does not have any CPU bottlenecks so does not need any extra optimization.

Example server-side code:

```
EdSharedSecret shared_secret;
if (!tun_server.ValidateChallenge(CHALLENGE, ip, port, &shared_secret))
      // Invalid CHALLENGE message
```

The server MUST ignore any invalid CHALLENGE message as if it was never received.

After the CHALLENGE has been validated, it is safe to allocate memory for the new client.

The server will then generate a KEY message and send it to the client.

Example server-side code:

```
u8 response[TUNNEL_KEY_BYTES];
MyTunnelServer.GenerateKeyResponse(&shared_secret, prng,
                                   &client->MyTunnelSession, response);

Send response, sizeof(response)
```

S2C KEY

After the client receives the KEY message, it validates and processes the KEY message by invoking the TunnelClient.ProcessKeyResponse() method.

Example client-side code:

```
if (!MyTunnelClient.ProcessKeyResponse(KEY))
      // Invalid KEY message
```

If the client receives an invalid KEY message it MUST ignore the message as if it were never received.

Handshake implementation details

Example Code

Definitely check out the test projects that are part of libcat to see how Tunnel can be applied.

Packetloss

It is up to you how to handle packetloss. I would recommend having the server cache its KEY message and respond to any additional CHALLENGE messages with the cached message until the client realizes it is connected and sends something other than CHALLENGE messages. A simpler and similarly secure alternative would be to start over at HELLO.

Magic Constants

You are encouraged to change all of the magic constants used in the protocol to personalize it. This is open-source software!

Client build

It is a good idea not to include server code in the client application. To remove the server code easily, simply comment out `INCLUDE_SERVER_CODE` ***in <cat/crypto/publickey/ECC.hpp>. This will require hackers to do much more work to reverse-engineer your application.***

Extra data in handshake packets

You are free to add more fields to the packets in the protocol outlined by this document. I would not recommend it unless you definitely know what you are doing, since it is very easy to make a mistake in this sort of protocol.

You CAN add a Session Message (see next section) to the end of the server's KEY message to send data as soon as the client connects, which would save one extra UDP/IP header.

Using TCP/IP instead of UDP/IP

This will work but the attacker would then have access to the TCP/IP header, which might be vulnerable to additional attacks outside of the scope of this documentation.

ICMP failures

To detect when the server is down faster, I recommend detecting ICMP errors before receiving the first COOKIE message from the server. Disabling ICMP after the first COOKIE message improves security.

For other details, please read the comments in the headers and other related documentation.

Session Encryption

After the handshake completes, messages to and from the server are encrypted and authenticated.

Both sides use the same packet structure but a different key to encrypt.

Example client-side code to package a message into a packet:

```
u8 response[TUNNEL_OVERHEAD + message_bytes];
(fill in message bytes here)
MyTunnelClient.Encrypt(response, message_bytes);
Send response, sizeof(response)
```

Notice that the client leaves TUNNEL_OVERHEAD bytes free for the encryption. The overhead is 6 bytes for both C2S and S2C session messages. 16 bits for the IV and 32 bits for the authentication hash.

Example client-side code to unpack a message from a packet:

```
if (!MyTunnelClient.Decrypt(SESSIONMESSAGE, total_bytes))
      // Invalid session message
// Message bytes are now available starting at buffer + TUNNEL_OVERHEAD
```

The client and server MUST ignore an invalid session message as if it were never received.

The server can perform the same operations on its TunnelSession object for each client.

Powers of the attacker

I am assuming that the attacker does not have physical access to the server, but he may have access to the client. The attacker is allowed to view, inject, drop, and modify all packets between the client and the server. He can measure the time each message is sent and received on each side, and he can delay the messages. Basically the attacker has the same powers as your ISP and/or government.

Potential Attacks and Countermeasures

Here I provide my argument for why this library achieves the conjectured security level. Potential problems you might care about are in ***red****.*

Key Agreement

Stealth protocol

By requiring the HELLO message to have contents unique to your protocol, the server will only respond to actual clients and not probes attempting to see what UDP ports are open, assuming that your server's firewall is set up properly.

Connection flood resistance

By operating statelessly until the client sends the COOKIE back in its CHALLENGE message, the server does not allocate any memory and does minimal processing if it is flooded with valid HELLO messages, and it sends the same amount or fewer bytes back to clients that send valid HELLO messages.

To flood the server with valid HELLO and CHALLENGE messages requires the client to send more data than the server, making flooding much more difficult. Furthermore, since the CHALLENGE messages must include the correct COOKIE, it means that all flooding clients are using their real IP addresses. So the server might automatically enter a new firewall rule to ignore packets from their IP address.

Another possibility is that an attacker controls part of the Internet infrastructure and can simulate control of an IP address (both spoof and listen to packets from that address). In this case, they would be able to simulate a flood to shut down a single client. Maybe the server should ignore floods from IP addresses of connected clients -- I am not sure what to do in that case.

Man-in-the-Middle (MitM)

As stated, this protocol is immune to MitM attacks. This is the standard guarantee provided by ECDH protocols so long as the server's public key is known to the client ahead of time.

Discard

The attacker can discard handshake packets, causing the connection to fail.

Replay

If the C2S HELLO is replayed, the worst that could happen is a valid S2C COOKIE. The attacker would just be making a new connection to the server like anyone else and no information is leaked.

If the S2C COOKIE is replayed, the client could get confused and fail to connect. But the attacker could also just cut the Ethernet cord and achieve the same thing.

If the C2S CHALLENGE is replayed and the embedded cookie is somehow valid, then the server would accept the client as a new connection using the same shared secret. The attacker wouldn't know the shared secret because he never learned the client's private key. This means that the KEY response could

not be decrypted by the client. And anyway, the SessionKey in the KEY response is chosen by the server and it is different every connection so no information is leaked.

If the S2C KEY is replayed then the client will start sending packets encrypted with some random key the attacker doesn't know. That would not help the attacker at all.

Delay

The attacker can cause the handshake to go more slowly. The worst that could happen is the client is unable to connect.

Side-Channel: Timing Attack

The ECC code performs a modular inversion with the Extended Euclidean algorithm that has runtime (very) loosely related to the secret key of the server. While I do not believe this to be a security hole given the powers of the attacker outlined above, you can definitely prevent it by adding random delay to the server KEY message response. I personally don't think this is a problem.

Other parts of the ECC code are hardened against timing attack. Though seriously the worst that could happen is an attacker knows a few bits are zero in the secret key. This would reduce the security level by less than 0.5 bits for each guessed bit of the secret key. So you might even want to remove the side channel protection for more speed.

Breaking the Asymmetric Cipher

The latest papers I have read indicate that the toughness of discrete logarithm for well chosen curves and base points is equivalent to brute forcing a key that is half the number of bits or harder. I believe that my curve parameters and random base points are well-chosen for security. See the next section for how these were chosen.

The Twisted Edwards curve I am using is equivalent to a traditional Weierstrass curve with no known extra weakness. The advantage is faster math.

Session Messages after the Key Agreement

Flood

Messages with invalid hashes will be ignored, so even if the attacker can spoof the client's IP address it should have little impact on the connection. Chance of guessing the correct hash is 1 in 2^32 or 0.000000023% of the time.

MitM

The messages are encrypted with a key that the attacker does not know at this point, so the worst he can do is discard, delay or replay messages.

Discard

The messages can be classified and discarded based on packet length.

For example, if a game cheater knows that all "you have died" packets are 73 bytes and the game protocol is poorly designed, then he can discard just those messages to gain an advantage. The Tunnel security layer does not provide any additional protection against these attacks. Only a well designed game protocol can do that.

Replay

If any session message is replayed from a different session then it will be using the wrong key, so the hash will filter it out.

This Tunnel does not prevent an attacker from replaying one of the last 32767 messages from the current session. This is not a new avenue of attack on the top-level protocol, just something that I haven't found an efficient counter for yet. Your message protocol will still need to counter this somehow. Note that legitimate users could still perform this attack even if I countered it.

Delay

Delaying messages from the client or server could be used to simulate latency, which may be beneficial if the top-level protocol has that weakness. Again this is not a new attack just something that my tunnel does not counter.

Side-Channel: Timing Attack

There are definitely no timing attacks against the ChaCha cipher.

Breaking the Symmetric Cipher

There is very little cryptanalysis of the ChaCha cipher, but it is considered to be one round harder to break than Salsa, which is a well-analyzed eSTREAM candidate, both by the same author. Salsa security at 12 rounds is 128 bits at worst, so ChaCha should provide 128 bits of security also. The advantage is slight speed and theoretical higher security.

Forward Secrecy

If the client's private key for a session is ever determined, then all packets in that session can be decrypted.

If the server's private key is ever determined, then all packets from all sessions with that server in the past can be decrypted after the key is broken. If there is a way to fix it without doing two scalar point multiplications on the server per connection I will implement that solution. EC-HMQV requires "2.5" scalar multiplications, which is undesirable.

Other attacks are currently unknown. Please let me know if you think of one!

Email: mrcatid@gmail.com

Appendix: Choosing Elliptic Curve Parameters

My ECC implementation (see http://catid.mechafetus.com/news/news.php?view=141) uses a twisted Edwards curve as defined by axx + yy = 1 + dxxyy over Fp. This curve was introduced by D. J. Bernstein, Peter Birkner, et al in their paper "Twisted Edwards Curves" (2008) [1].

I choose a = -1 for slightly faster addition and doubling formulas, based on the Extended Projective coordinates introduced by H. Hisil, K. Wong, et al in their paper "Twisted Edwards Curves Revisited" (ASIACRYPT 2008) [2].

Before using these formulas the field prime, d, and base point need to be selected.

Choosing p:

I chose the field prime p = 2^256-c, where c is a small number that makes p prime = 3 (mod 4). I chose this form since for my code this is the most efficient form to use apart from Mersenne primes 2^52 1-1 and 2^127-1, which are of the wrong size. It enables me to perform square root by modular exponentiation in my own code, rather than using someone else's library for a more complex square root algorithm.

To pick c = 189, I just iteratively tried all the p = 2^256-c = 3 (mod 4) starting from c = 1, until the Rabin-Miller primality test said it was a prime.

The most common operation for point multiplication is modular reduction after a multiplication or squaring. For this special form of p, the modulus operation can be computed in less than half the time of a Montgomery Reduce operation typically used for RSA encryption, so Montgomery RNS form is unnecessary. This is a good thing since using RNS makes the code larger and more complex.

Choosing d:

The choice of d requires exhaustive computer search. I have been using a modified copy of the MIRACL library's version of the SEA algorithm for point counting. In short, my modifications allow Mike Scott's code to find values of d for which the number of points on the curve is some large prime number x4.

MIRACL's SEA code accepts any Weierstrass curve to check for point count. Since Twisted Edwards curves can all be written in this form, I can use the existing SEA code so long as I first convert Twisted Edwards parameters a, d to Weierstrass parameters a4, a6.

I am using the mapping given in [1] to go first from Twisted Edwards to Montgomery form (Byy = xxx + Axx + x):

A = 2(a+d)/(a-d)
B = 3/(a-d)

Then from Montgomery form to Weierstrass, as given by 13.2.3.c in the "Handbook of Elliptic of

Hyperelliptic Curve Cryptography" (2006):

a4 = 1/(BB) - AA/(3BB)
a6 = -AAA/(27BBB) - Aa4/(3B)

Once it is in the familiar Weierstrass form, SEA happily can perform point counting for that choice of d.

After I look at small values of d, the one that has the largest point count will win. Finding a few values of d that work is actually not hard and can be done in an hour on a modern Intel quad-core Xeon processor. Just starting four parallel searches improves the speed nicely... =)

Some choices for d:

For curve: -X^2 + Y^2 = d*X^2*Y^2 + 1 over Fp, p = 2^256-189, NP = number of points on the curve

NP/4=28948022309329048855892746252171976963449087812878447882682178384830544116457 for d=321 <= chosen
NP/4=28948022309329048855892746252171976963364137860193714257833303761363762534599 for d=3567
NP/4=28948022309329048855892746252171976963356671321081301322781900858624717722023 for d=2087
NP/4=28948022309329048855892746252171976963209239424227502814070775171816900352269 for d=4238
NP/4=28948022309329048855892746252171976963194848828905762674632598658951271884767 for d=889
NP/4=28948022309329048855892746252171976963193685037190250905176734116833185051207 for d=2715
NP/4=28948022309329048855892746252171976963171131370530754794550345168351098815157 for d=3272
NOTE: These are all prime and very close to p/4. Note that the cofactor is 4 in each case.

Furthermore, d=321 is not a square in Fp, and a=-1 is a square in Fp, which are the conditions noted in [2] for the completeness of the unified addition and doubling formulas.

For curve: -X^2 + Y^2 = d*X^2*Y^2 + 1 over Fp, p = 2^384-317, NP = number of points on the curve

NP/4=985050154909861980306976002503590345126993481761636166698988985967401709086492052322341742906124734454422110449361 3 for d=2857 <= chosen
NP/4=985050154909861980306976002503590345126993481761636166698981636785782425223731290120557909148421038748618157970382 1 for d=2161
NP/4=985050154909861980306976002503590345126993481761636166698656178944137717269177071376046589923502132761346950995315 1 for d=394
These are also all prime and very close to p/4. The cofactor is also 4.

For curve: -X^2 + Y^2 = d*X^2*Y^2 + 1 over Fp, p = 2^512-569, NP = number of points on the curve

NP/4=3351951982485649274893506249551461531869841455148098344430890360930441007518428505811628503485002357654150053918764604945126948031916092776575822477898019, d=3042
NP/4=3351951982485649274893506249551461531869841455148098344430890360930441007518351768221124561641986955855153248518812921556717848912561407873860520202964569, d=3858
NP/4=3351951982485649274893506249551461531869841455148098344430890360930441007518349490051623856836233261142646242322171503970713904582841833043621026033442323, d=497
These are also all prime and very close to p/4. The cofactor is also 4.

Here is the modified MIRACL code used to generate these values of d:
http://catid.mechafetus.com/code/MIRACL_SEA_VS2008.zip

The changes are just in how sea.c interprets the command line parameters. README.TXT describes how to use it.

sea.c checks for the MOV condition and anomalous curves also, so these values of d pass these tests.

Choosing a base point:

I am using the cofactor method to avoid small subgroup attacks: After generating a random point on the curve, I multiply it by 4, which is the cofactor of NP as shown in the previous section. The result is used as the base point and becomes part of the public key of the server.

My code generates a new random base point whenever it chooses a new server private key, so the base point becomes part of the public key and makes the public key twice as large as it would need to be otherwise. I do not know if there is a security disadvantage to using the same base point for every server, but I personally do not mind the overhead so until I know for sure the public keys will be 128 bytes.