What's new in OpenSSH?

Damien Miller <djm@openssh.com>

Introduction

OpenSSH is approaching 12 years old

Still adding features

We don't do “big splash” releases, so you may not have heard of what has been developed recently

Let's fix that...

What's new?

Match

PKCS#11 support

sftp extensions

SSH protocol 1 deprecation

Certificate authentication

Elliptic Curve cryptography

Match

This feature isn't so new, actually introduced in 4.4

Allows variation of sshd's config depending on

- Username
- Group
- Source address
- Source hostname (if you trust DNS)

Replaces previous hack of multiple instances + AllowUsers/AllowGroups

Match

Very useful for:

Restricted accounts (e.g. mandatory chroot)

Limiting authentication by source network
(e.g disable password auth from internet)

Match – example: controlling auth

```
PasswordAuthentication no

# Override main configuration
Match address 10.0.0.0/8
  PasswordAuthentication yes
```

Match – example: controlling options

```
AllowTcpForwarding no

Match group wheel,fwd
  AllowTcpForwarding yes

# Wildcard predicates == ok
Match user hosted-*
  PasswordAuthentication no
  PubkeyAuthentication yes
```

Match – example: anonymous sftp

```
Match user anonsftp
    ForceCommand internal-sftp -R
    ChrootDirectory /chroot/home
    PermitEmptyPasswords yes
    PasswordAuthentication yes
    AllowAgentForwaring no
    AllowTcpForwarding no
    X11Forwarding no
```

PKCS#11

PKCS#11 is a standard for cryptographic tokens

- Smartcards
- Hardware Security Modules (HSM)

Key storage in PKCS#11 devices appeared in 5.4

- Deprecating the old opensc/libsectok smartcard code

Smartcards can store authentication and CA keys

Match - example: use a smarcard key

```
ssh -I /opt/lib/mycard-pkcs11.so \
    user@host
```

ssh(1) will dlopen() the specified PKCS#11 provider and use it to enumerate and use keys on the device it supports

Match – example: smartcard w/ agent

```
# add keys
ssh-add -s /opt/lib/mycard-pkcs11.so

# remove keys
ssh-add -e /opt/lib/mycard-pkcs11.so
```

PKCS#11 – future work

Currently, using a smartcard means trying *all* its keys

- Nice if we could select better
- We can to some extent using IdentityFile in ssh(1), but this isn't obvious

Would be nice to allow host keys to be stored in PKCS#11

- Root compromise != persistent hostkey theft

No support for certificates (discussed later)

sftp extensions

The sftp protocol as OpenSSH implements is actually "sftp v.3", or more verbosely "draft-ietf-secsh-filexfer-02.txt"

sftp is a good example of how consensus standards development can produce bad protocols

Original versions had an elegant simplicity

- Basically the Unix file API as protocol methods
- Open file => handle
- Read from handle => data
- Reminiscent of 9p from Plan 9

sftp extensions

Later versions (upto v.6) accreted features

- “Text mode” files to better support Windows
- “Record mode” files to better support OpenVMS
- MIME types, Win32ish ACLs, byte-range locking

We think that we already have a do-everything network filesystem protocol in NFS, we don't need another

So we stopped implementing features after sftp v.3

This was unfortunate for people who needed new features

sftp extensions

Fortunately, the sftp protocol has some extension mechanisms that we have started to use

In the initial protocol "hello" message, client and server can advertise extensions that they support

When the protocol is established, named extension methods can be used

- E.g. "statvfs@openssh.com"
- Downside: named extensions are more bandwidth hungry than numbered protocol methods.

sftp extensions

We have added a number of extensions already:

- posix-rename@openssh.com
  - Standard sftp v.3 uses a link()+rm() raceless rename
  - Use as "rename" via sftp(1) in OpenSSH >= 4.8
- statvfs@openssh.com, fstatvfs@openssh.com
  - Use as "df" via sftp(1) in OpenSSH >= 5.1
- hardlink@openssh.com
  - Use as "ln" via sftp(1) in OpenSSH >= 5.7

sftp extensions

More to come:

- user/group names for files
  - sftp v.3 only support numeric uid/gid
- fsync() file handle method
- O_NOFOLLOW open mode

Some of these are very useful for user-space filesystems that use the sftp protocol

- (though people must realise it is inherently racy)

Deprecation of SSHv1

We recently completed a staged deprecation of SSHv1

Why?

- SSHv1 lacks many features of SSHv2
- SSHv1 offers no viable extension mechanism
- SSHv1 suffers from a number of unfixable cryptographic weaknesses

SSHv1 – CORE-SDI "SSH insertion attack"

Found by CORE SDI in 1998

Fundamental problem is SSHv1's use of CRC as a integrity code

- CRC is linear; changes in its input lead to predictable changes in its output

CORE SDI figured out how to inject data by calculating how to reconstruct a valid CRC

Attack cannot be prevented, but can be probabilistically detected

- Detection code was buggy too!

SSHv1 – Use of MD5 in the protocol

SSHv1 uses MD5 for key derivation and RSA public key authentication

No way to specify a different algorithm

MD5 is broken as a cryptographic hash

- Attackable for the RSA authentication case?

SSHv1 – downgrade attack

If a client and server support SSHv1 and SSHv2, a man-in-the-middle may silently downgrade their connection to SSHv1

SSH advertises supported versions in initial banner:

- e.g. "**`SSH-1.99-OpenSSH_5.8`**"
- SSHv2 checks banners, SSHv1 does not

Attacker can modify banners, force use of SSHv1

- Why? Attack vulnerable code or protocol components.

SSHv1 – even more crypto badness

Not-quite PFS (ephemeral host key)

Probably vulnerable to CPNI-957037 "Plaintext Recovery Attack Against SSH", Information Security Group at Royal Holloway, U. London, November 2008

Weak private key file format

SSHv1 - weaknesses

- We deprecated it in two steps
  - 4.7 – new server installations no longer enable SSHv1
  - 5.4 – client must be explicitly configured to use SSHv1
- Quite a few people still liked SSHv1 because of speed
  - It's easy to be fast when you are insecure :)
  - Why is SSHv2 slower? MAC and key exchange
  - We implemented a fast MAC (umac-64) and now a fast key exchange (ECDH)

Certificate authentication for OpenSSH

A new, very lightweight certificate format (not X.509)

Released in OpenSSH-5.4, improved in OpenSSH-5.6

Design goals: simplicity, modest flexibility, minimal attack surface

Why (another certificate format)

"We have OpenPGP and X.509, why reinvent the wheel?"

- OpenSSH will not accept complex certificate decoding in the pre-authentication path:
  - PGP and X.509 (especially) are syntactically and semantically complex
  - Too much attack surface in the pre-authentication phase
  - Bitter experience has taught us not to trust ASN.1

Differences from X.509

X.509	**OpenSSH**
hierarchical CA structure	no hierarchy (maybe later)
complex identity structure	identity is just a string
multi-purpose	SSH auth only
identity bound by key owner	identity bound by CA
complex encoding	simple encoding
infinitely extensible	extensible enough (I think)

OpenSSH certificate contents

So, what is in an OpenSSH certificate?

```
Nonce
Public key (DSA or RSA)
Certificate type (User or Host)
Key identifier
List of valid principals
Validity time range
Critical options
Extensions
Reserved field (currently ignored)
CA key
Signature
```

Critical options and extensions

Options that limit or affect certificate validity or use. May be "critical" or not. Critical = server refuses authorisation if it doesn't recognise an option

Present options are basically a mapping of `.ssh/authorized_keys` options into the certificate:

Critical	**Non-critical**
`force-command`	`permit-X11-forwarding`
`source-address`	`permit-agent-forwarding`
	`permit-user-rc`
	`permit-pty`
	`permit-port-forwarding`

Generally, options that *grant* privilege are non-critical.

Certificate encoding

The certificate is encoded using SSH-style wire primitives and signed using SSH-style RSA/DSA signatures

- Very little new code.
- Minimises incremental attack surface

Fixed format: all fields must be present (though some subfields are optional) and must appear in order.

Certificates are extensible using new critical options, extensions, new types (in addition to user/host), or the currently-ignored "reserved" field.

Certificate integration - User auth

User authentication can be trusted CA keys listed in `~/.ssh/authorized_keys` or via a sshd-wide trust anchor specified in `sshd_config's` *TrustedCAKeys* option

Principal names in the cert must match the local account name in the case of `authorized_keys.` A `principals="..."` key option allows some indirection here.

For certs signed by *TrustedCAKeys*, an optional *AuthorizedPrincipalsFile* (e.g. `~/.ssh/authorized_principals`) allows listing of certificate principals to accept.

Certificate integration - host auth

CAs trusted to sign host certificates must be listed in a *known_hosts* file (either the system `/etc/ssh/known_hosts`, or the per-user `~/.ssh/known_hosts`)

Trust of host CA's can be restricted to a specific list of domain wildcards:

```
@cert-authority localhost,*.mindrot.org,*.djm.net.au ssh-rsa AAAAB3Nz...
```

Fallback: If a host presents a cert from an unrecognised CA, then it is treated as a raw public key for authentication purposes (normal key learning rules apply)

Certificate integration - CA operations

CA operations are built into *ssh-keygen*:

```
# Create a keypair
$ ssh-keygen -qt ecdsa -C '' -f ~/.ssh/id_ecdsa -N ''

# (on the CA) sign the public key to create a user cert
$ sudo ssh-keygen -s /etc/ssh/ssh_ca_key \
    -I "djm" -n djm,shared-nethack \
    -O source-address=10.0.0.0/8 \
    -O force-command=/usr/bin/nethack \
    -O permit-pty \
    -V -1d:+52w1d id_ecdsa.pub \
    -z 314159265
Signed user key id_ecdsa-cert.pub: id "djm" serial 314159265
for djm,shared-nethack valid from 2011-02-10T14:28:00 to 2012-
02-09T14:28:00
```

Certificates - Revocation

Revocation story is kind of weak at present.

Emphasis is on making certs short-lived rather than revocation

User authentication keys can be revoked using a flat file of public keys. Host keys are revoked in known_hosts.

Revoked keys print a scary warning on use:

```
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@       WARNING: REVOKED HOST KEY DETECTED!               @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
The RSA-CERT host key for localhost is marked as revoked.
This could mean that a stolen key is being used to
impersonate this host.
```

Can do an OCSP-like protocol if necessary in the future. (patches welcome)

Certificates - Future plans

Write a HOWTO-style document

Improve revocation - OCSP-like protocol?

Ability to store OpenSSH certs in X.509 certs for easy smartcard use

- OCTET-STRING certificate extension under an OID IETF allocated to the OpenSSH project

Combined OpenSSH and X.509 CA tool: sign CSR and OpenSSH pubkey in a single operation

Maybe implement chained certificates

Elliptic curve cryptography

OpenSSH 5.7 introduced Elliptic Curve Cryptographic key exchange and public key types

- Key Exchange is ECDH
- New public key type is ECDSA

Implemented according to RFC5656 “Elliptic Curve Algorithm Integration in the Secure Shell Transport Layer” by Douglas Stebila

What is ECC?

Elliptic Curve Cryptography (ECC) is public-key cryptography calculated using Elliptic Curves over finite fields

- Contrast with traditional public key algorithms that usually calculate in a finite integer group

Elliptic curves over finite fields provide an algebraic group structure in which the discrete logarithm problem is “hard”

- Discrete Log problem (DLP): Given g^x, find x
- Solving the DLP is harder in curve fields than in prime fields, so key lengths can be shorter

What is ECC?

Since the DLP is hard, DLP-dependent cryptosystems work

- DSA => ECDSA
- DH => ECDA
- ECRSA isn't common since RSA doesn't rely on the DLP

Since key lengths are shorter, ECC-based algorithms are usually faster for a given security level

More introductory information at:

- http://wikipedia.org/wiki/Elliptic_curve_cryptography
- http://imperialviolet.org/2010/04/ecc.html

ECC in OpenSSH: Key Exchange

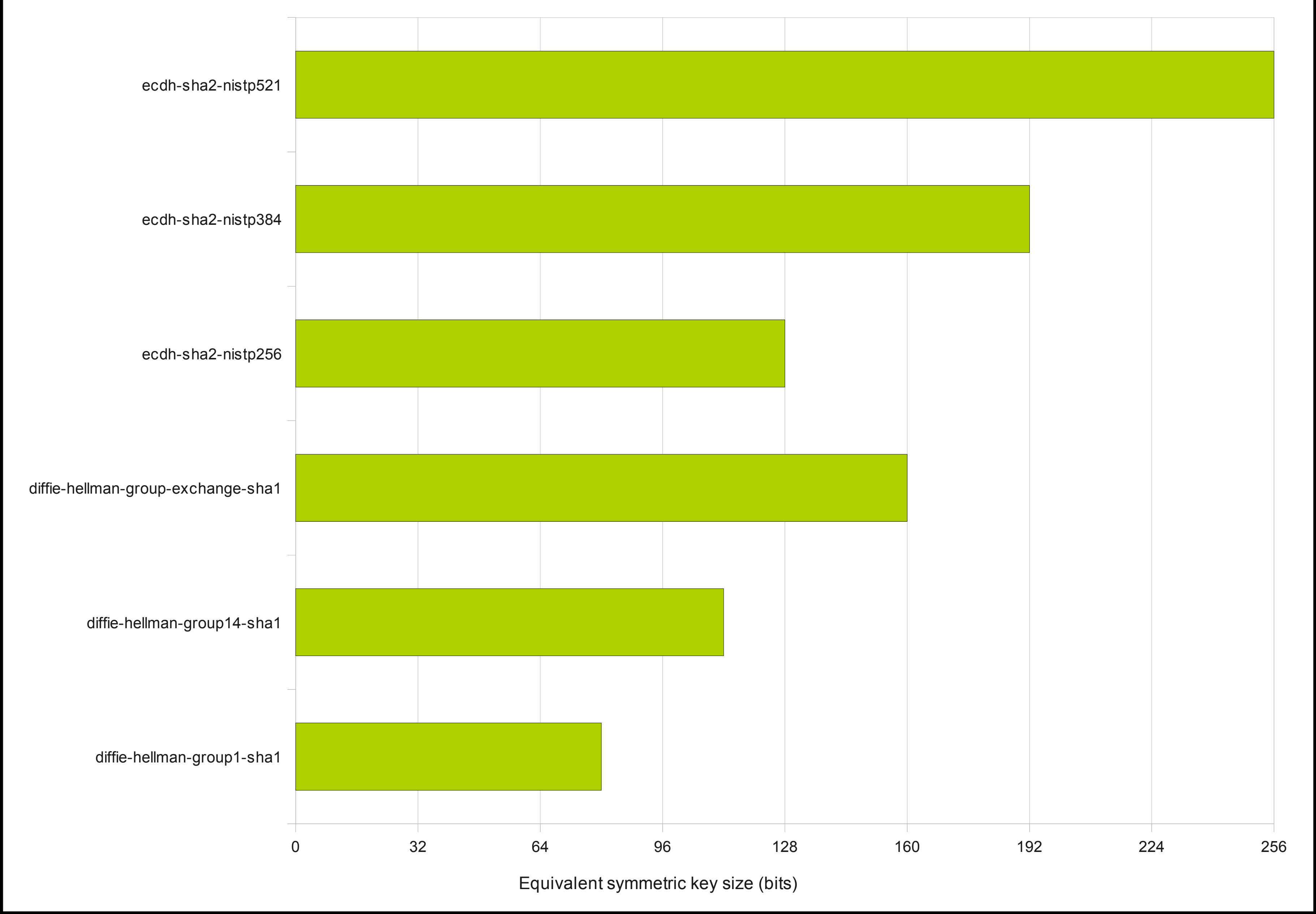
OpenSSH >= 5.7 support Elliptic Curve Diffie-Hellman key exchange (ECDH)

Three security levels provided by three different protocol methods, each with its own curve field:

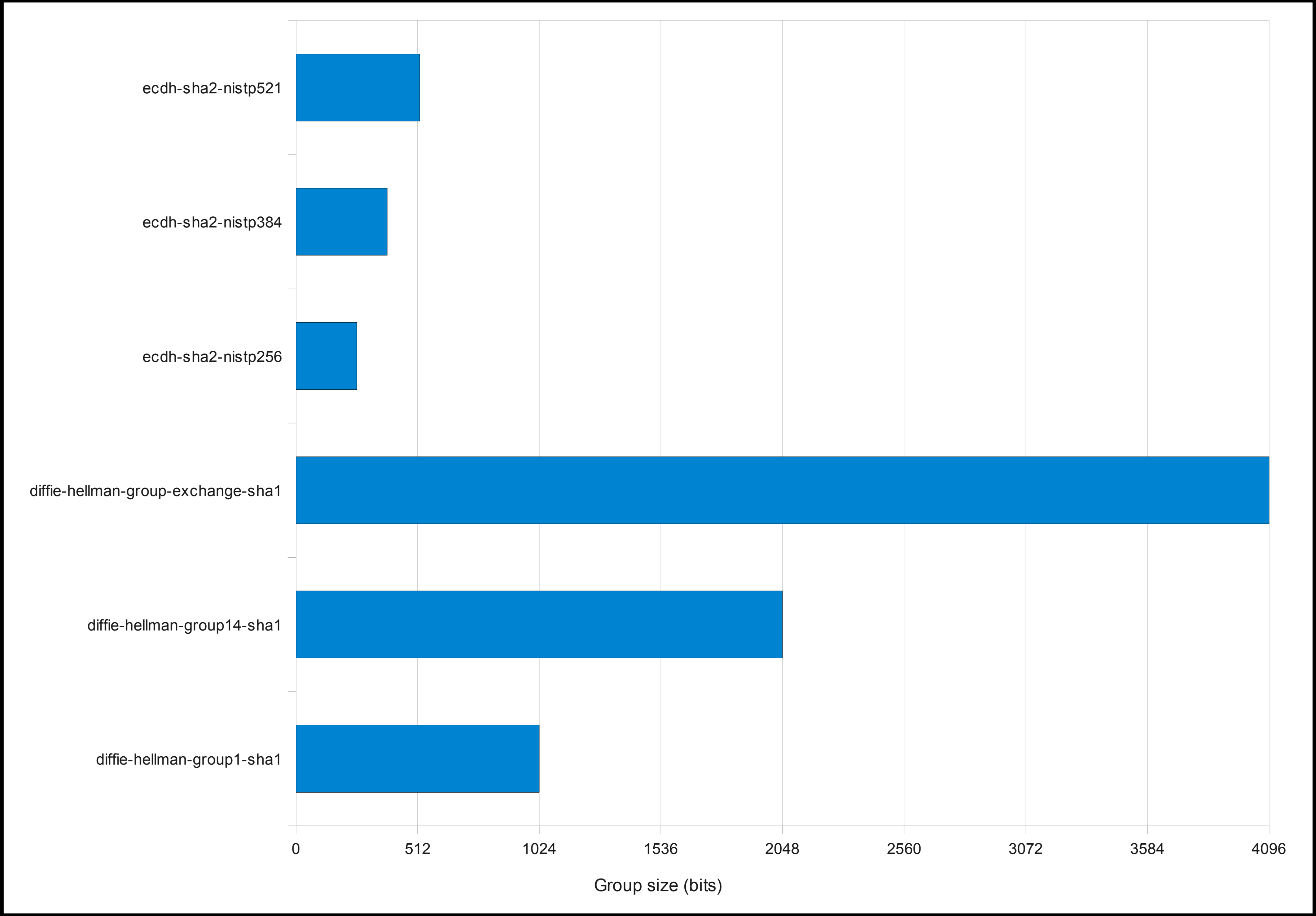
- ecdh-sha2-nistp256
- ecdh-sha2-nistp384
- ecdh-sha2-nistp521 (note: not 512)

OpenSSL is used for elliptic curve operations, including point serialisation/parsing

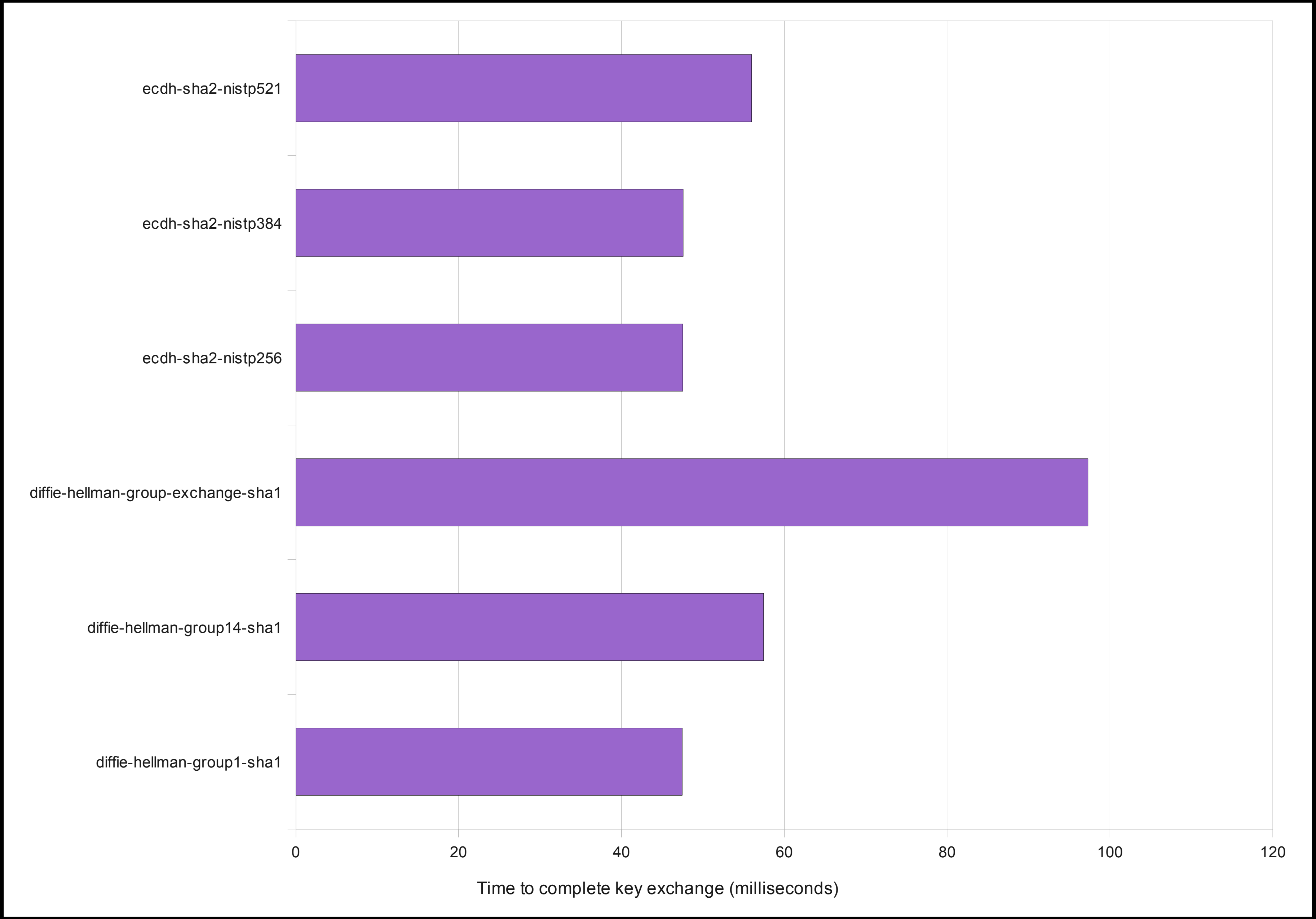
KEX – symmetric equivalent security



KEX - group size (bits)



KEX - time to complete



ECDH key exchange

On by default if both ends support it in OpenSSH >= 5.7

If you require more than 128 bits of symmetric equivalent security, then you should use the `sshd_config` *KexAlgorithms* option to chose the 384 or 521 bit ECC curve field.

OpenSSL's ECC implementation is still being optimised

- 2 x speedup in -current
- Possibly 4 x speedup if we use a hand-optimised 224-bit curve field.

ECC in OpenSSH: keys

OpenSSH >= 5.7 supports Elliptic Curve DSA (ECDSA) for user and host keys

Again, the curve field is explicit and defines the security level of the algorithm

We use the same three curves (mandatory in RFC5656):

- ecdsa-sha2-nistp256
- ecdsa-sha2-nistp384
- ecdsa-sha2-nistp521

All hidden behind “ecdh” key type used on command-line

ECC in OpenSSH: keys

ECDSA is slightly faster than regular DSA

- Still a benefit in symmetric-equivalent security
- Shorter keys too

ECDSA keys can appear wherever RSA or DSA keys work:

- User keys (`~/.ssh/id_ecdsa`, `~/.ssh/authorized_keys`)
- Host keys (`/etc/ssh/ssh_host_ecdsa_key`)
- Certificates (as signed key or as CA)

ECDSA keys are preferred when both ends support them

OpenSSH - what's next?

- Small features
- Refactoring
- Better testing
- More bugfixes

Thanks