

Sodium

crypto library

Table of Contents

Introduction	1.1
Installation	1.2
Quickstart and FAQ	1.3
Projects using libsodium	1.4
Commercial support	1.5
Bindings for other languages	1.6
Usage	1.7
Helpers	1.8
Padding	1.9
Secure memory	1.10
Generating random data	1.11
Secret-key cryptography	1.12
Authenticated encryption	1.12.1
Encrypted streams and file encryption	1.12.2
Encrypting a set of related messages	1.12.3
Authentication	1.12.4
AEAD constructions	1.12.5
ChaCha20-Poly1305	1.12.5.1
Original ChaCha20-Poly1305 construction	1.12.5.1.1
IETF ChaCha20-Poly1305 construction	1.12.5.1.2
XChaCha20-Poly1305 construction	1.12.5.1.3
AES256-GCM	1.12.5.2
AES256-GCM with precomputation	1.12.5.2.1
Public-key cryptography	1.13
Authenticated encryption	1.13.1
Public-key signatures	1.13.2
Sealed boxes	1.13.3
Hashing	1.14
Generic hashing	1.14.1
Short-input hashing	1.14.2

Password hashing	1.15
The pwhash* API	1.15.1
Key derivation	1.16
Key exchange	1.17
Advanced	1.18
SHA-2	1.18.1
HMAC-SHA-2	1.18.2
The Scrypt function	1.18.3
Point*scalar multiplicaton	1.18.4
One-time authentication	1.18.5
Stream ciphers	1.18.6
ChaCha20	1.18.6.1
XChaCha20	1.18.6.2
Salsa20	1.18.6.3
XSalsa20	1.18.6.4
Ed25519 to Curve25519	1.18.7
Finite field arithmetic	1.18.8
Ristretto	1.18.8.1
Custom RNG	1.18.9
Internals	1.19
Roadmap	1.20

The Sodium crypto library (libsodium)

Sodium is a modern, easy-to-use software library for encryption, decryption, signatures, password hashing and more.

It is a portable, cross-compilable, installable, packageable fork of [NaCl](#), with a compatible API, and an extended API to improve usability even further.

Its goal is to provide all of the core operations needed to build higher-level cryptographic tools.

Sodium is cross-platforms and cross-languages. It runs on a variety of compilers and operating systems, including Windows (with MinGW or Visual Studio, x86 and x86_64), iOS and Android. Javascript and WebAssembly versions are also available and are fully supported. Bindings for all common programming languages are available and well-supported.

The design choices emphasize security and ease of use. But despite the emphasis on high security, primitives are faster across-the-board than most implementations.

[Version 1.0.18](#) was released on May 31, 2019.

Downloading libsodium

- [Github repository](#)
- [Tarballs and pre-compiled binaries](#)
- [Documentation](#)

Mailing list

A mailing-list is available to discuss libsodium.

In order to join, just send a random mail to `sodium-subscribe {at} pureftpd {dot} org`.

Offline documentation

This documentation can be downloaded as a PDF file here:

<https://www.gitbook.com/book/jedisct1/libsodium/details>

License

[ISC license](#).

See the `LICENSE` file for details.

Thanks!

The development of libsodium is entirely made by volunteers. We would like to specially thank the following companies and organizations for their contribution:

- [Paragonie Initiative Enterprise](#), who donated a Raspberry Pi to ensure that the library works perfectly on this hardware. Thanks!
- [Private Internet Access](#), who sponsored a [complete security audit](#). This is amazing, thanks!
- [Maximilian Blochberger](#) and Joshua Small, who both generously donated \$100. This will help a lot to cover the infrastructure costs. Thanks again, Max and Joshua!

People who designed the primitives and wrote implementations the library is based on can be found in the [AUTHORS](#) file. This project wouldn't exist without them.

Also a huge "thank you" to people and companies who contributed bindings for their favorite programming languages. A list can be found in the [THANKS](#) file.

Another huge "thank you" to package maintainers who have been doing an amazing job at building packages for many distributions and operating systems.

Finally, thanks to **you** for reading this documentation and for the awesome projects you are going to build with this library!

Installation

Compilation on Unix-like systems

Sodium is a shared library with a machine-independent set of headers, so that it can easily be used by 3rd party projects.

The library is built using autotools, making it easy to package.

Installation is trivial, and both compilation and testing can take advantage of multiple CPU cores.

Download a [tarball of libsodium](#), preferably the latest `stable` version, then follow the ritual:

```
./configure
make && make check
sudo make install
```

Since different files are compiled for different CPU classes, and to prevent unwanted optimizations, link-time optimization (LTO) should not be used.

On Linux, if the process hangs at the `make check` step, your system PRG may not have been properly seeded. Please refer to the notes in the "Usage" section for ways to address this.

Also on Linux, like any manually installed library, running the `ldconfig` command is required in order to make the dynamic linker aware of the new library.

Compilation on Windows

Compilation on Windows is usually not required, as pre-built libraries for MinGW and Visual Studio are available (see below).

However, if you want to compile it yourself, start by cloning the [stable branch](#) from the Git repository.

Visual Studio solutions can be then found in the `builds/msvc` directory.

In order to compile with MingW, run either `./dist-build/msys2-win32.sh` or `./dist-build/msys2-win64.sh` for Win32 or x64 targets.

Pre-built libraries

[Pre-built x86 and x86_64 libraries for Visual Studio 2010, 2012, 2013, 2015, 2017 and 2019](#) with `stable` additions (see below) are available, as well as pre-built libraries for MinGW32 and MinGW64.

They include header files, as well as static (`.LIB`) and shared (`.DLL`) libraries for all the supported compiler versions.

Note for Visual Studio

Projects willing to statically link Sodium must define a macro named `SODIUM_STATIC` . This will prevent symbol definitions from being referenced with `__declspec` .

Cross-compiling

The library can be cross-compiled. This is an example of cross-compiling to ARM using the GNU tools for ARM embedded processors:

```
export PATH=/path/to/gcc-arm-none-eabi/bin:$PATH
export LDFLAGS='--specs=nosys.specs'
export CFLAGS='-Os'
./configure --host=arm-none-eabi --prefix=/install/path
make install
```

`make check` can also build the test apps, but these have to be run on the native platform.

Note: `--specs=nosys.specs` is only required for the ARM compilation toolchain.

Please note that using libsodium on ARM Cortex M0, M3 and M4 CPUs is not recommended if side-channels are a concern.

Compiling with CompCert

Releases can be compiled using the CompCert compiler.

A typical command-line to compile Sodium on a little endian system with CompCert is:

```
$ env CC=ccomp CFLAGS="-O2 -fstruct-passing -Usize_t" ./configure && \
make check && sudo make install
```

Stable branch

We recommend using distribution tarballs over cloning the [libsodium git repository](#), especially since tarballs do not require dependencies such as libtool and autotools.

However, if cloning a git repository happens to be more convenient, the [stable](#) branch always contains the latest stable release of libsodium, plus minor patches that will be part of the next version, as well as critical security fixes while new packages including them are being prepared.

Code in the `stable` branch also includes generated files, and does not require the autotools (libtool, autoconf, automake) to be present.

To check out the stable branch, use:

```
git clone https://github.com/jedisct1/libsodium --branch stable
```

Tarballs of the `stable` code are also available for download, and are recommended if you are compiling libsodium from source.

Getting started

See the [quickstart](#) and [usage](#) sections to get started!

Integrity checking

Distribution files can be verified with [Minisign](#) and the following Ed25519 key:

```
RWQf6LRCGA9i53mlyec04IzT51TGPpvWucNSch1CBM0QTaLn73Y7GF03
```

The `.minisig` file should be present in the same directory as the main file. The typical command to verify a file is:

```
minisign -P RWQf6LRCGA9i53mlyec04IzT51TGPpvWucNSch1CBM0QTaLn73Y7GF03 -m <file>
```

Or with GnuPG and the following RSA key:

```
-----BEGIN PGP PUBLIC KEY BLOCK-----  
Version: GnuPG v1 (OpenBSD)  
  
mQINBFTZ0A8BEAD2/BeYhJpEJDADNu0z5E08E0SIj5VeQdb9WLh6tBe37KrJJy7+  
FBFnsd/ahfsqoLmr/IUE3+ZeJNJ6QVozUKUAbds1LnKh8ejX/QegMrTgb+F2Zs83
```


8ju4k6GtWquW50miG7+b5t8R/oHlPs/1nHbk7jkQqLkYAYswRmKld1rqrrLFV8fH
SAsnTkgeNxpX8W4MJR22yEwxb/k9grQTxnKHHkjJInoP6VnGRR+wmXL/7xeyUg6r
EVmTaqEoZA2LiSaxaJ1c8+5c7oJ3zSBUveJA587KsCp56xUKcwm2IFJnC34WiBDn
K0LB7lNxiT3BnnzabF2m+5602qWRbyMME2YZmcISQzjiVkt8062qmKfFr5u9B8Tx
iYpS0a19HvZqih8C7u/SkeGzb0NfbmmJgFuA15LVwt7I5Xx7565+kWeoDgKPlfrL
7zPrCQqS1a75MB+W/fOHhCRJ3IqFc+dT1F4hb8AAKWrERVq27LEJzmxXH36kMbB+
eQg336JlS6Tmqe1VFb15PgtcFh972jJK8u/vpHY0EBPij5chjYQ2nCBmFLT504UZ
Y4Gm8Z3QLFG1Ee0iz+uRdnfchxwflkjng1UhKXSq5yu0AAeMaNoYftCf1hAHG6tx
vWYIijRxDu5c8cDZsKMuLQ3406DuvPZyeCy6q8BTfW18miMMhIH0QTS9MwARAQAB
tC5GcmFuayBEZW5pcyAoSmVkaS9TZWN0b3Igt25lKSA8akBwdXJlZnRwZC5vcmc+
iQI2BBMBCAAgAhsDAh4BAheABQJU2dF6BAsJCacFFQoJCAsFFgIDAQAACgkQIQYn
qrwn+Gp0BAakJu5yZhlPBIZnDZMr0oJ/pJiSea7GUCY4fVuFUKLpLlSjIaSxC4E
2oWG8cJoMdMhww1x166rRZPdXfPw8eC5r+h8m25HBj649FjMUPDi2r9uQgPdBy80
I+gFlrsinSy7xbd1USpjrcYYCx9jYjjTwH6L1QZa+YCMFya8dob/NcdzQ0o7cNRu
5NG988cScssscXYXzI6SMouSwPGCMrQHAsM31Yb8YfbJLuDxFRCZY5+qiR8DXDzW4
Lp68fJq0X/UGW9Q+i29LMTvZZWDGBQ9bwQNTvDrPZ8SYp249cM0sR4W7FK4Y00ea
YRTBfcXaeXEKAP1ZqYrY22BDiHJ05IGY72D3j3vPATAYigwjR/qNF0t/DaERFpQ4
L7RD+E6WLHATFWxZH/APck6q8bY4Ehr8GJWA77sIQn/Ctvap759QKB8nrerT6lA
0cojhS5Ie8Lro6YsMAXDqwjzsv+VgnTgql8oAFmuU+o+6cmHUwGNHgEs+xe2UDQi
kxu685g0CHfHmBwue391glHufQdveChy5eikif6q6Ndg7VH9mR335o8VJ9I+Vp/k
3W8XZBA90Euwrxyj1EzWvcb2WGxRUHVZ32w+E9CICvFFV7JiTntG3t1Ch4/bbFwr
wdkc5EZTh0c6B7YfIkEwnOnBovWBPEBkSGve371MsqBuKuBr1W4jegyIRgQQEQgA
BgUCVnNRHAACRCSa8UXHN6k0WxZAKCGlk6DvVCqExkBd60EsaEo0BgH5ACfcVQa
z/FEgCdRsJeLi7xNwZXZ2200IUzyYw5rIERlmlzIDxnaXRodWJAcHVyZWZ0cGQu
b3JnPokCNgQTAQgAIAIbAwIeAQIXgAUCVNnRaQQLCQgHBRUKCQgLBRYCAwEAAAOJ
ECEGJ6q6cJ/hslQIAI2l+uRlwmofiSHo/H2cUDN02Nn7uRfcVIw9EItTmdU6KKx9
nkgFP3Y3lUwLQFP6aQhQJyHBU5QGqn9n8k8+jEPciTL7hcbTuY0YRuz0mp9bJ8r
ruqGxTrZuogvIVntwnn1HvgAbu13HKu+3K0LYDmWqosVNf0a8GjHj10ZDuNDPQVb
X6NWDes+jLdeUsxVKUZHlOC3CiRCSHJzz3G1g09QU78LQAFCIID07G07xPjqbvEX
nsys5f120LXB4NqBlIamEdyztv+CwIZBM9Ni6ytPnEhwzTHzHwi95oNa+AtpUlgG
RYjYtMR9pxCqVkrplwrwhA4dbS07HLixQIRa57F1/5LwKRR4e7IGhntPzow8hr8y
qg4AAVCZqr5aB82L0JAMP6ZlC7kQb9/YxGYw4Vwf6qCY8Iw74MvIL5wW0zSv/orB
entHeP0Z/Ozx3UXKA2chNElEWbZ9e0IZBXgcj/JDfK8e0VTqv1ItHlM2ZkvCbyhV
fER8I8AHPnfzkwXvWFeDKem08rakqDeNQ3h4BeiCBCVHPeSudIWSG3oC01guy9/h
xMJR2yAwik+35sCcZbrgTTN0oQepRMuZ34niIBK0jUh7t1M5sBMNgxEAIeKjJf64
DEudNz+xUgek5N+BXx7hryuVC3s1y6H42zt0jPtPpHVUw98gwpv5V7QRLBs0iEYE
EBEIAAYFAITZ0RwACgkQkmvFFxzepDn8sACdF51BycwRvMpkFPea1Yi3/B1EOs0A
oJT9afe3zQn0lcIuBFBzpd0TsecUtCZGcmFuayBEZW5pcyA8ZnJhbmsuZGVuaXNA
Y29ycC5vdmguY29tPokCNgQTAQgAIAIbAwIeAQIXgAUCVNnRegQLCQgHBRUKCQgLB
RYCAwEAAAOJECEGJ6q6cJ/h0LgP+wfcw2SCFvD7sFlnmD6oJNP+ddtt+qbxDGXo
UbhrS1N88k6YiFRZQ+Z84ge9RgQXA74xuWlx8g1YBESq01rYCGQ4C+Ph+oUO+a3X
k+wmEzINnjCF8CQzZQ3vdXvWmshKzqC2yyeR235WC/BSHsqsr+TRFEMGa68ju8s7
UF8ZQaBzbM0ttUtrc0UqhnS16xV5lH9gBkVbMWIN1pAeJcFRL6MB92Vv5tWjayua
w76vxmwPhu6quUlwNynVYBg5kpBjqMOLHaX1x+SA5F6aI6E3kqxeYurwV6Ty+/
FIns+Awl+IFPey5ctwS0XkizhtqxpMNHau9resNRjneIjNVTL0N1uaxvmpJttMd/
CdTXh+guxDBfH6Vr9nmExy2qbiHDJ06Sm874UYtnBZdB7Fi0cNF1DlEZKaZyYaLw
RA/TeLi2IaIdkRFLsaFdo144nfceZ2fra2Q0830w6uShNZZAhu0ZVEKLVt/VJqCL
6hts7vhKuCBcnlpoNOZptRPJf8RMLh4qwtNiZadDcm16TpvkyTQUAWH+GvTML0UR
5sLH0tZ7MUaHO/c5UWQWJ0muaw0KgdKLi3ixZtGbNNDc9F7wRoObUH70m/0s5IRy
no058ofDCmurPDP+10e0QawtgVz2nFXcFF0qTw4H6L/sXlzbm27HuqEHuYrZpTl/
Njn0chjBiEYEEBEIAAYFAITZ0RwACgkQkmvFFxzepDnrmQCfdaiJcQsAZaSfEf01
VxZaY0KEVf0An1xVULYvo5M4sta0tILFu3UthzBGtDdGcmFuayBEZW5pcyAoSmVka
S9TZWN0b3Igt25lKSA8MGRhewRpZ2VzdBwdXJlZnRwZC5vcmc+iQI2BBMBCAAg

BQJU2dKRAhsDBAsJCACFFQoJCAsFFgIDAQACHgECF4AACgkQIQYnqrpwn+FqRxAA
wWm+f6mo9nCoGRD4r4jrSLuJ5ApyIXRQ3L5DL/MeITRMPNDps00pvKIIGmGv19n5
Ani7uf0cnQLkTVj1179U5BTnahk2fDS0Cx1FyslpR9A7tX6qQMtIyBE4cdPhjVue
Z0wI+PfJSleFFmPQ3ESlBkZeNGJqBQINSbpo9qMhhyYRZy/Fk4k0QzAdXpa63kPX
1KVoTsvz1902frLim7QY8oTI8Vbij0CB+HfhHuLmolc039/S47hF+5ygERK5Fwjo
mSx+Q2fKx9P35TZqQ9Zw73e3gS9YUeRT4LU7ZwdmulftfCaVLmIUx4GUDPasmNbA
WLpKHEwLln0YJ00kIzD+2q2zclzUmGgdgGcEUwLb6vpWLJ41MsmHknZg0zm/yG6/
sasA0jU1wKxeRLHeSxnh3PYb+v36kHXsRViqPlwxe9PGmLK9p9wD0yS/dk2LsJbE
1hnUZfw7114VdivrL567My/0sG3SbIUb/DxHuVkgHU9LHHlca4z5VmFc7v2+sc0+
6IczFW86FKI8m+q8zLhHcquKgZpumxvwjEoAbjl9123bqZKm1e8pHL3bTQa6bSv9
isNsw3T9eHeEB7fRbL1Y0ZjvMQuYLF82t2tu+E4xbUYZZrmlRYGwBGFUBRprtJ0e
XeUvxFgAnazyNNXxxH03PMiCxpCp0e7+x64fKVPmFu5Ag0EVNnQDwEQAMnv/UG9
7vAtIyeG+lPalmhnl0NQ07I4Rz+vigZHAx08t7QYh0Y0YLZFj1m011f81c5X1oxV
7dKwh+sHMQJ3fk0mQbG6VGRlMRTAPk45GsaRcAnczNzCZWw0s4f92ybc9Th4dNR8
dUk90t+tFitPGnFHGHmjwUYMc7u8BN191/SNiJipxuHjUR1hXQE+RXrlgkow9S8I
bisHytd5IC0XGz337coYkdJLzx1Adp0MGN4n5qymLrhjBivv2a/R+mweUAD7I18I
Ynj581a1rp2kLmnoJacL0R9R2ZbSjDBevKpitmy3kqHS59vChw80asBRwr10++Ea
V0LnWdKKbc1U809RP1Ac0l66KjKj3mmiQQKdpb2oHHD0uJsx84kqC0kowdQf12wR
stygySAC8CJXnsAKThdDvsQTKMX6WKg4wtSJw0ELRtNCQZzH8iE6eq9MXZijvG6H
j9WyZ2L2ee00bKn0uEDGvpPMLwCF0jCxL32x/Jr95sqAt2p0DcBFH5d4jK7tqHQ
YzNwt8ibbbG1wzRFTgq/5igV+n9q9P/h8bwQHjUJyqbJyJuwt4l/oTSTKZ5bZ0IAR
KS/+Y/Y9b/BBXRzRP/D1Lha0ndH43E6HmEWGS2PhUUPn3V6TQz0q5npaTXKhq/f8
XMYEqvbQ3qjFREa+LLgmFLAWD7rc8h2WYVp7ABEBAAGJA8EGAEIAAKFA1TZ0A8C
GwwACgkQIQYnqrpwn+GCvHAAsc00PYCRzcgDwDW0rT3g5yi8dt3NmDGL9c6/ohKV
waWSIDlwFtbZNiZ/fr91VCdDfhUSohtn6E7XvKYdVNO4NRLIbSgRc7Y/C4P+91Eh
k+6mlXy1Eil/GN6YXBsQvDSz1xw+Csz3Y6kq2m1xiSHFuZrP0PS75x+vIAKbIsPa
uu5IyEh/wAW1vY/pnzS7TJtY2r8Qsv/5xt+zUd1GB0ZJq7IZ/1GveltRMJrfhcCT
KPQRWdMv0aEioeBwYAM8sc9UrrePM9jSpT3uCYwuJl1d4M94+tqt7tqvkr6dluXF
+4WweuPXo65jSBl094BEfT5dVbt0TqmG6eTgnPggh1j7PpIghyqUU0v8YP15DUnZ
UuHz14CEcQWNUeQ+xK9N2/nflaq8R4LPDJjupSWIw5tZv8NWj+EA/zyxggX+q2pr
3qld+IU0n8cR/RT1LvZ9L5t1fvTqjPgDqXJIremih0bLOGEV0+0xWEa0850VzyU
QTt2EBHzSxHkC0CED6CgR8l48YGsKJrHCju0vQ+lgVtAkgyBeVFeFhrKa242TmVB
NlZCKs25wUhGhWbLv334p+KTG4d79J+iKYbh8n0C/gBK0YzDX3gLbL+6wes0xYia
WSRBfx9hfPCfFLDGGSy7yViH8YcOGig6IV9+DWBCSy0Z0d0IXWNvTLF+3d1BFD4
dlG5Ag0EVNnQNWQANZNOFI4cM9TYFCM0YIiH1UaXoibNE7kZ1qDM/06y5HTU0Sn
m2koCYMTqtVaigAq/tXiUJLBzoHwh17CzDx5L3/ISHMhdqWAFccUZII2NW/XEEH7
knwnqn5tki2CZCzFE+GxTUm7M7fBW2pgPvVt/Ord+DhmEKP0A+fdKHS3x/EUn8Vs
vJoYEkxg9fT14eqYk+oALFxm6vW9UAF00VZ/J0XzeDTux0+6p6NqjcykKeG5GixA
dHpRopfekSLQx3sZqfFBEhuiIX7P1lAQxHpPqKcPG82avQ5x9tvZ2RVdk/55hcK
gNhdcbDGWqkNENb0vTmom2a/gDNgb7pf12jJa9t2RRVC8oyYh+zVftLhf2G1wMVv
vwuX01U2A0/1UQ7K33t6lQ2mEmbudyFJCso3kIJ598efTw2ZPkeEKZ+adsIBQbd
CSEm0B/S+DS8CDTLTfS5nN5T3rGn07LzPf983uP9CLb0Dyt05dqF1Hl+4XicMT3P
Qtz1T+P7X7nPQL9FUw0WUBHqfYhNsnV17m6M/ODoKsyjd192nj0xvyD6zVaffcx
2zX+SYEaIIiDFhVfprhwTuruk0fax3nNTLd1JeiraUejSNCn60VxTsp203Y0H8
quLtvswF6V5lr57WQxGQxQmS5JQV9wreYzuA339ApUqukfWmhiPDHbQVWAe3ABEB
AAGJBD4EGAEIAAKFA1TZ0DcCGWICKQkQIQYnqrpwn+HBXSAEGQEIAAYFA1TZ0DcA
CgkQYvJbWstvdTq1jg/8Dm6BicjEbcNphwpsjj0uoPB49I0fKFxSM2uUh6PI+wtc
LtikJsNyGvXdm7oGE/uXiki5S++91pZ5oTV931HVzP8e4vip5IRCCwFk6NisRmiZ
nN/xMejLnK3s51pmK5UJhoYymrETGiUKj1uu5BqewRXZ4wWH2kzIusBzIc537shR
Gqk+LgwY7/x4aKY+5Z46VpAGS104a6WdWt1RLZz0z0x+tPIrAYo0f72hdHg2enZE
rqkhi90dy/5hCsaJRl+raEZVDSgg0t00hmtNLSWAX3YPINp1qSqvn5EQk8FhZuh
RaonpXg0wZLc82oIYEZ0KnhJ7HBgV/jf781I5ZPdk9m22GbASwkiJwNmzfAhGEPu
/NX3iweDPfU4ULb0vejs3ivQTE0rF47u3ps/6S0rBXS7f23ZBw7nwYryezCeQUV8

```
RCKkk+xUPv5YU0DpGtViDrfxeucXW8W05V0BsCfpa2PTXvj4VjP6UGRUcX3SVTcA
VnvKA mfsDa/4+4A0EvfgQFRzuex8tthFbPW2pLJEQPpVFuxAK0foUhw78HFL7NRV
TFx3jUWgGAM7PA9FI9h1rrU5dXyi8uXwBjaXcEaIts7WE0NGjFzEbub6kJldryh1
5ZCMkmOcBU7SkSmI95b0JwvYdGGiEc04eh7ci4p0FH0ZNqKfpjyfpTgtFgS5Ldne
pBAA8ubnR6+b7gGa0Qk/rR0TYHoSq9GXVAqhbmY69lfsXQ9EXoiAzNZnhJLtj1J7
86Z3Bgd9X+MXrrPoJLVGmBTT8yT337KY/+rbk16E5oL1eItnsJ0xgprD1gkWUNaa
pRXLKdA86ogoU8sE/9Wr2CN6dCdPCmjmcmWwGHY5V6lMf3NPIAQbS4izuU/w+IE
gPnBo45BPkxP2Hyvho0ek+pxpsqL8uLQzuIjtwgWvM0ocVQrpBNr6kQ99hvr8feY
6k0I5MoGsagW3R65m7DAfz/x1o03QmWT/kg2dcwqiEbZL3phX1QpQtdJk05+JTYQ
F0WP5sPzQ7DaIP7Mo2Njhqvn05NR9/kEzX1yEQck3BI4vKNHsiAQ1/J94uiu9Aze
w6ddP04Ax7LycK0W0eNVNAT6a3tFJbQrve3ZoDDSNXAa70VKmpdrsrwnX+/4+rly
Z7lj7rnMWCe9jllfZ2Mi+nIYXCrvhVh0t70HVGwpSq28B/e2AFsQZxXcT4Y+6po7
aJADVdb+Ll0AuF6xB3syle1Im0iADCW9UAWub1oi0r9jv0+mHEyc3kaF0kPU5zK0
I9cg891jc0BV/qRv89ubSHifw9hTZB0dDjXzBjNwNjBHqkYDaLsf1izeYHEG4gEO
sjoMDQMqgw6KyZ++6FgAUGX5I1dBOYLJoonhOH/lNmXjQvc= =Hkmu
-----END PGP PUBLIC KEY BLOCK-----
```

Reporting vulnerabilities

We encourage users and researchers to use PGP-encrypted emails to transmit confidential details regarding possible vulnerabilities in the Sodium library.

Details should be sent to: j [at] pureftpd [dot] org using the PGP key above.

Quickstart / Libsodium FAQ

Boilerplate

A project using libsodium should include the `sodium.h` header. Including individual headers is neither required nor recommended.

The `sodium_init()` function should then be called before any other function. It is safe to call `sodium_init()` multiple times, or from different threads; it will immediately return `1` without doing anything if the library had already been initialized.

```
#include <sodium.h>

int main(void)
{
    if (sodium_init() < 0) {
        /* panic! the library couldn't be initialized, it is not safe to use */
    }
    return 0;
}
```

Is libsodium cross-platform?

Yes. A message can be encrypted in Python on a MIPS CPU, decrypted in Javascript using Chrome on Windows, and its signature can then be verified by an iPhone app written in Swift.

The available algorithms are the same on all supported platforms. Input and output formats are also identical on all platforms. And bindings for various programming languages tend to leverage the original algorithms and formats.

How do I generate random numbers that are safe to use for cryptography?

Use the `randombytes` API.

How do I compute a hash?

Use the `crypto_generichash` [API](#).

What is the difference between a secret key and a password?

A password has a couple requirements to fulfill:

- It has to be reasonably short
- It must be easy to type on any keyboard
- Users may have to remember it

As a result, the number of possible passwords of a given size is fairly small. For the same size, the number of distinct passwords actually used is even way smaller, and their distribution is skewed. Even if mitigations exist, guessing (brute-forcing) a password is often practical.

A secret key, on the other hand, is a sequence of bits that can be of any size, doesn't have any restrictions, and is generated by software or specialized hardware. A 32 bytes key represents 256 bits, none of them being predictable by observing other bits from that key or from previous keys.

Secret keys are not designed to be typed on a keyboard. This is binary data. They are stored as files, kept in secure memory or sent over the network. They can be unique, and offer far more security than passwords.

Libsodium almost exclusively uses secret keys.

If an application really needs to use a password as a secret key, it should call the `crypto_pwhash()` function first. This computes a secret key from a password, using an intentionally CPU-intensive and memory-hard function (to slow down brute-force attacks).

But native secret keys using the `*_keygen()` function should always be preferred.

How do I encrypt data?

One-shot encryption, where everything fits in memory

- Create a secret key using `crypto_secretbox_keygen()`
- Create a nonce using `randombytes_buf(nonce, sizeof nonce)`
- Use `crypto_secretbox_easy()` to encrypt the message, and send/store the resulting ciphertext along with the nonce. Unlike the key, the nonce doesn't have to be secret.
- Use `crypto_secretbox_open_easy()` to decrypt the ciphertext using the same key and

nonce.

If everything doesn't fit in memory, or is not available as a single chunk

- With libsodium 1.0.14 and beyond, use the `crypto_secretstream` API.
- Otherwise, read the guide to [encrypting a set of messages](#)

How do I safely store and later verify a password?

- Use the `crypto_pwhash_str()` and `crypto_pwhash_str_verify()` functions, described in the [password hashing guide](#).

How do I encrypt a file using a password?

- Derive an encryption key from the password using `crypto_pwhash()`.
- Use that key with the `crypto_secretstream` API.
- File metadata should probably be part of the encrypted data, or, if it is not secret, included as additional data.

How can A and B securely communicate without a pre-shared secret key?

Use the [key exchange API](#):

- A and B both call `crypto_kx_keypair()` to create their own key pair. Secret keys have to remain secret, but A can send its public key to B or even make it available to everyone. The same applies to B's public key.
- A uses `crypto_kx_client_session_keys()` along with B's public key and its key pair to create a set of shared keys to communicate with B.
- B uses `crypto_kx_server_session_keys()` along with A's public key and its key pair to create a set of shared keys to communicate with A.

The shared keys computed by A and B will be identical. There are two of them. One can be used to encrypt and decrypt message in one direction (from A to B) and the other one to encrypt and decrypt messages in the other direction (from B to A).

To actually encrypt and decrypt data using one of these shared secret keys, use one of `crypto_secretbox_*()` , `crypto_secretstream_*()` or `crypto_aead_*()` .

How can I derive multiple keys from a single master key? Like what HKDF does?

Use the [key derivation API](#).

Do I need to add a signature to encrypted messages to detect if they have been tampered with?

No. Signatures are designed to allow non-secret data to be verified by many parties, using a public key. For examples, a signature can be used to verify the authenticity of a firmware update.

When `A` encrypts a message for `B` using a shared secret key using `crypto_box()` , `crypto_secretbox()` , `crypto_seal()` , `crypto_secretstream()` or `crypto_aead()` , an authentication tag is also computed, and should be sent to `B` along with the encrypted payload.

During the decryption process, the secret key is used to check that the authentication tag is valid for the given encrypted message. If that message has been modified, the tag will not be valid, and decryption functions will return an error code. Knowing the secret key is required in order to create a valid tag, therefore only `A` and `B` can create such a tag.

How do I hide the length of a message?

Use [padding](#).

Shall I call `crypto_generichash_blake2b` or just `crypto_generichash` ?

Always use the high-level API if one is available. The low-level API it is based on is guaranteed not to change before a major revision of the library. And if a high-level API needs to use a different construction, it will expose a different set of functions.

This is true for all APIs provided by the library.

Why is `crypto_stream()` barely documented and not even present in some bindings?

The `crypto_stream()` API generates a deterministic sequence of bytes from a seed, and optionally applies the XOR operation between that sequence and some input sequence.

Performing the XOR operation once produces content that resembles random data, and performing the same operation twice restores the initial input sequence.

This can be seen as a form of encryption. However:

- If an adversary replaces the ciphertext with nul bytes, what will be decrypted is the original, complete sequence derived from the seed. This can have catastrophic implications with some (badly designed) protocols.
- More generally, anyone can modify the ciphertext without this being detected. Since a stream cipher is XOR'd with a message, targeted bits can be flipped. Knowing the secret key is not required.

If a deterministic sequence has to be derived from a seed, for example for unit testing, libsodium provides the `randombytes_buf_deterministic()` function.

For actual encryption, other options such as `crypto_secretbox`, `crypto_aead` and `crypto_secretstream` must generally be used over `crypto_stream`, as they will add and verify an authentication tag to detect data that has been corrupted or tampered with.

`crypto_stream()` is only useful as a building block to design custom constructions. As-is, it is completely insecure.

I want to write bindings for my favorite language, where should I start?

Start with the `crypto_generichash` and with the `crypto_secretstream` APIs. These are the trickiest to implement bindings for, and will provide good insights about how to design your bindings.

Libsodium users

Applications using libsodium

Some applications using libsodium. Send a [pull request](#) to add yours to that list.

- [Aspia Remote Desktop](#): A remote desktop implementation (client and server) for Windows systems.
- [Apache Tuweni](#): A set of libraries and other tools to aid development of blockchain and other decentralized software in Java and other JVM languages.
- [Babble Browser Extension](#): a platform agnostic browser extension that allows for easy encryption and decryption of text data across the web.
- [Cloaker](#): Very simple cross-platform file encryption.
- [CurveLock](#): Message encryption for Windows.
- [Cyph](#): End-to-end encrypted, authenticated, and ephemeral chat with voice + video + file transfers.
- [DAT](#): Nonprofit-backed data sharing protocol for applications of the future.
- [Detsign](#): Use passphrases to generate deterministic ED25519 signing keys.
- [Discord](#): All-in-one voice and text chat for gamers that's free, secure, and works on both your desktop and phone.
- [Dircifrar](#): A directory synchronization and encryption tool.
- [Dovecot](#): Dovecot is an open source IMAP and POP3 email server for Linux/UNIX-like systems, written with security primarily in mind.
- [Duniter](#): Duniter is an open source Relative Money Theory distributed forged blockchain and clients (silkaj, duniterpy, cesium, ...) uses libsodium
- [emberclear](#): An Open Source, privacy-first, encrypted chat app using progressive web app technologies.
Built with [Ember](#) using all of the latest experimental features.
- [EQEmu](#): An Open Source project committed to providing custom Everquest servers.
- [Evernym Plenum](#): A byzantine fault tolerant protocol.
- [Fastd](#): A fast and secure tunnelling daemon.
- [FeCl](#): A secure message-passing server based on libsodium.
- [Flocksy](#): An anonymous file synchronisation tool.
- [Glorytun](#): A small, simple and very fast VPN.
- [Ghost-Notes](#): a user-friendly, password protected, note taking application for iOS.
- [Habitat](#): Habitat, by Chef, is automation that travels with the app.
- [Hat-Backup](#): A backend-agnostic snapshotting backup system.
- [KadNode](#): A small P2P DNS resolution daemon based on a Distributed Hash Table.

- [Keystone Enclave](#): An open source secure enclave for RISC-V processors.
- [KeepassXC](#): KeePass Cross-Platform Community Edition. KeePassXC stores your passwords safely and auto-types them into your everyday websites and applications.
- [Kickpass](#): A stupid simple password manager.
- [Kryptonite](#): The new home for your SSH private key on your iPhone or Android.
- [Lageant](#): libsodium Authentication Agent.
- [Lawncipher](#): An embedded, encrypted, multi-purpose document store.
- [libagentcrypt](#): Library for symmetric encryption using SSH Agent. It includes a command line utility able to encrypt/decrypt files and/or passwords securely with your ssh keys
- [Lumimaja](#): PasswordSafe with Argon2 KDF, data encrypted with ChaCha20Poly1305, and Yubikey support.
- [MLVPN](#): A multi-link VPN (ADSL/SDSL/xDSL/Network aggregator)
- [Magic Wormhole](#): Get things from one computer to another, safely.
- [MaidSafe](#): A new Secure way to access a world of existing apps where the security of your data is put above all else.
- [Metastream](#): Watch streaming media with friends.
- [Minisign](#): A dead simple tool to sign files and verify signatures.
- [Molch](#): An implementation of the Axolotl ratchet.
- [Muon](#): A fast stateless VPN with obfuscation to escape traffic monitoring.
- [NTPsec](#): A security-hardened implementation of Network Time Protocol Version 4.
- [NanoChat](#): A P2P, E2E encrypted and discoverable chat application on top of nanomsg library.
- [Network Next](#): A real-time CDN for game traffic. *You* control the network!
- [Open Reputation](#): An open source decentralized platform that maps identity and reputation onto the Internet-of-things.
- [OpenBazaar](#): A free market for all. No fees, no restrictions.
- [OpenGKS](#): An RFID gate keeper solution that automatically tracks, records data and sends text messages (SMS) and email notifications to parents upon a student's entrance and exit in school.
- [OpenR](#): Facebook's internally designed and developed routing protocol/platform.
- [PAVE](#): The Password Manager. Easy password sharing for teams. No cloud.
- [PCP](#): Pretty Curved Privacy (pcp1) is a commandline utility which can be used to encrypt files.
- [Pbox](#): A CLI password manager.
- [Petmail](#): A secure communication and file-sharing system.
- [Pichi](#): An Application Layer Proxy controlled via RESTful APIs.
- [PipeSocks](#): A pipe-like SOCKS5 tunnel system.
- [PowerDNS](#): PowerDNS has been designed to serve both the needs of small installations by being easy to setup, as well as for serving very large query volumes on large numbers of domains. Additionally, PowerDNS offers very high domain resolution

performance.

- [QtCrypt](#): Lightweight, portable application that encrypts files and directories using a symmetric-key algorithm.
- [RavenDB](#): A linq enabled document database for .NET.
- [Rebel Backup](#): Encrypted backups of important files to Dropbox and Google Drive.
- [Remembear](#): a full-featured, multi-platform password manager, by TunnelBear.
- [Reop](#): Reasonable expectation of privacy.
- [Rubinius](#): Rubinius is a platform for building programming language technology.
- [SODA](#): The SODA project aims to investigate the relationship between server components and server performance.
- [SaltStack](#): SaltStack software orchestrates the build and ongoing management of any modern infrastructure.
- [Sandstorm](#): An open source operating system for personal and private clouds.
- [ShadowSocks](#): A secure socks5 proxy, designed to protect your Internet traffic.
- [Secrets](#): Easily and securely store passwords, credit card numbers, bank details, and confidential information. Available for Mac and iOS.
- [Simple DnsCrypt](#): A simple management tool for dnscrypt-proxy.
- [Snackis](#): A post-modern enigma-device
- [SPAKE2+EE](#): A SPAKE2+ Elligator Edition implementation for libsodium.
- [Splonebox](#): An open source network assessment tool with focus on modularity.
- [Stellar](#): An open platform for building financial products that connect people everywhere.
- [TREES](#): A Dovecot email storage encryption plugin.
- [Tbak](#): Encrypted, compressed, distributed backups.
- [Telehash](#): An embeddable private network stack for mobile, web, and devices.
- [Tezos](#): a new decentralized blockchain that governs itself by establishing a true digital commonwealth.
- [Tinfoil Chat](#): an onion-routed, endpoint secure messaging system.
- [Tox](#): A new kind of instant messaging.
- [Ultrapowa Clash Server](#): UCS is a server emulator for the famous game Clash of Clans
- [VOLTTRON](#): VOLTTRON is an innovative distributed control and sensing software platform. Its source code has been released, making it possible for researchers and others to use this tool to build applications for more efficiently managing energy use among appliances and devices, including heating, ventilation and air conditioning (HVAC) systems, lighting, electric vehicles and others.
- [Wifibroadcast](#): Transmitter and receiver of UDP packets using raw WiFi radio.
- [Wire](#): Modern, private communications. Crystal clear voice, video and group chats. No advertising. Your data, always encrypted.
- [Wireguard-rs](#): Rust implementation of Wireguard.
- [Zbox](#): zero-details, privacy-focused embeddable file system.
- [Zcash](#): Zcash is a decentralized and open source cryptocurrency that aims to set a new

standard for privacy through the use of groundbreaking cryptography.

- [libsodium for Universal Windows Platform](#): a C++ Windows Runtime Component for UWP applications.
- [libsodium password hashing schemes for Dovecot](#): Dovecot plugin to support Scrypt and Argon2 for password hashing.
- [Wordpress](#) uses libsodium for verified updates.
- [C-lightning](#), a [Lightning Network](#) implementation in C, uses libsodium.
- [SilentDragon](#) + [SilentDragonLite](#) : GUI full node + lite wallets for the [Hush](#) cryptocoin, which is focused on secure communications.

Libraries using libsodium

Some libraries and frameworks using libsodium. Send a [pull request](#) to add yours to that list.

- [Asio Sodium Socket](#): A header-only C++14 library implementing custom transport encryption using libsodium and Asio's stackless coroutines.
- [Blobcrypt](#): Authenticated encryption for streams and arbitrary large files.
- [Cordova Minisodium](#): A minimal sodium plugin for Cordova - for iOS and Android.
- [EdCert](#): A Rust crate to sign and verify content using Ed25519.
- [Folly](#): An open-source C++ library developed and used at Facebook.
- [Halite](#): High-level cryptography interface for PHP.
- [Hyperledger Indy](#): Everything needed to build applications that interact with an Indy distributed identity ledger.
- [Innovault](#): A toolkit to store sensitive data from web forms in an encrypted database.
- [LibSQL](#): SQL authentication library.
- [MEGA SDK](#): SDK by mega.nz, a secure cloud storage provider that protects your data, thanks to end-to-end encryption.
- [Macaroons](#): Macaroons are flexible authorization credentials that support decentralized delegation, attenuation, and verification.
- [MatrixSSL](#): Lightweight embedded SSL/TLS implementation that can use libsodium as a crypto provider.
- [Minimal-TLS](#): A minimal implementation of TLS 1.3 in Rust.
- [MySQL-Sodium](#): MySQL UDF bindings for libsodium.
- [NaclKeys](#): Library to generate libsodium-net compatible KeyPair's.
- [Netcode.io](#): A simple protocol for creating secure client/server connections over UDP.
- [Noise-C](#): Noise-C is a plain C implementation of the Noise Protocol, intended as a reference implementation.
- [OAuthSDK](#): OAuth 1 and OAuth 2 framework in Swift for iOS.
- [Posh-Sodium](#): A powershell module.
- [Pull-box-stream](#): One way streaming encryption for Javascript.

- [Smart Encryption](#): Secure by default encryption for .NET
- [Sodium native](#): Low level NodeJS bindings for libsodium.
- [Stream Cryptor](#): Stream encryption & decryption with libsodium and protobuf for .NET
- [Tanker](#): an encryption SDK for JavaScript.
- [The Update Framework](#): A plug-and-play library for securing a software updater.
- [Yojimbo](#): A network library for client/server games with dedicated servers.
- [ZeroMQ](#): Connect your code in any language, on any platform.
- [libotr4](#): An OTR version 4 implementation.
- [libsaxolotl](#): An Axolotol implementation in C using libsodium.
- [libsodium-Laravel](#): Laravel integration.
- [libsodium-UE4](#): An easy to use cryptography plugin for Unreal Engine 4 based on libsodium.
- [minisign-net](#): .NET library to handle and create minisign signatures.
- [neuropil](#): An IoT secure messaging library with end-to-end encryption written in C and using libsodium
- [pgsodium](#): Postgres extension wrapper around libsodium.
- [spake2+ee](#): A SPAKE2+ implementation.

Companies using libsodium

Some companies using libsodium, possibly in non-opensource products. Send a [pull request](#) to add yours to that list.

- [Arpa Information Technology Solutions](#)
- [Assa Abloy](#)
- [Auth0](#)
- [Bearch Inc.](#)
- [Digital Ocean](#)
- [EAM GmbH \(bytejail\)](#)
- [Espressif Systems](#)
- [Facebook](#)
- [Infoblox](#)
- [Informatica](#)
- [Innovis Solutions KG](#)
- [Keybase](#)
- [Krypt.co](#)
- [Magento \(Adobe\)](#)
- [Malwarebytes](#)
- [Mullvad VPN](#)
- [OVH](#)

- [Paragon Initiative Enterprises](#)
- [Peerio](#)
- [Pi-lar](#)
- [Private Internet Access](#)
- [Q-Leap Networks](#)
- [Riseup](#)
- [Supercell](#)
- [Tanker](#)
- [Valve](#)
- [Wire](#)
- [Yandex](#)

Commercial support for libsodium

The following companies offer professional support services for libsodium and applications using libsodium:

Edge Security (<https://www.edgesecurity.com>)

Libsodium-specific services include:

- Cryptographic and implementation auditing of libsodium usage.
- Security review of constructions, protocols, and applications using libsodium.
- Consulting on libsodium usage in applications -- web apps, backends, mobile, embedded, kernel, etc.
- General consulting and review of cryptography inquiries.
- Implementation security and helping to work toward a vulnerability-free codebase.
- Reverse engineering of complicated binaries.
- General consulting, advising, mentoring, and development.

Paragon Initiative Enterprises (<https://paragonie.com>)

Libsodium-specific services include:

- Library integration, with a focus on web applications (PHP, .NET, Python).
- Bespoke or standard high-level protocol design and implementation, e.g. Noise.
- Security audits, for in-house library integrations and/or high-level protocols.
- Custom application development that requires cryptography.
- Consulting and mentoring services.

(Please submit a pull request if you want your company added to that list)

Libsodium and programming languages

Programming languages whose standard library includes support for libsodium

- PHP ≥ 7.2
- HHVM ≥ 3.20
- [Citrine](#)
- [Factor](#) ≥ 0.98

Bindings programming languages

- .NET: [libsodium-net](#)
- .NET: [libsodium-core](#)
- .NET: [nsec](#)
- .NET: [nitralibsodium](#)
- .NET: [Andhell.Crypto](#)
- Ada: [libsodium-ada](#)
- C++: [sodiumpp](#)
- C++: [tears](#)
- C++11: [sodium-wrapper](#)
- Chicken: [chicken-sodium](#)
- Clojure: [caesium](#)
- Clojure: [naclj](#)
- Common LISP: [cl-sodium](#)
- Crystal: [Sodium](#)
- D: [Chloride](#)
- D: [LibsodiumD](#)
- D: [Shaker](#)
- D: [Sodium](#)
- Dart (Flutter): [Flutter-Sodium](#)
- Delphi/FreePascal: [Delphi/FreePascal](#)
- Dylan: [libsodium-dylan](#)
- Elixir: [Salty](#)
- Elixir: [Savory](#)
- Erlang: [ENaCl](#)

- Erlang: [Erlang-NaCl](#)
- Erlang: [Erlang-libsodium](#)
- Erlang: [Salt](#)
- Fortran: [Fortium](#)
- Go: [libsodium-go](#)
- Go: [Sodium](#)
- Guile: [Guile-NaCl](#)
- Haskell: [haskell-libsodium](#)
- Haskell: [Saltine](#)
- Haskell: [Lithium](#)
- HaXe: [haxe_libsodium](#)
- HaXe (bindings for the Neko VM): [neko-libsodium](#)
- Idris: [Idris-Sodium](#)
- Idris: [Sodium-Idris](#)
- Java (Java Native Access): [libsodium-jna](#)
- Java (Android): [Lazysodium for Android](#)
- Java (Android): [Libstodium](#)
- Java (Android): [Robosodium](#)
- Java (Android): [libsodium-JNI](#)
- Java: [Apache Tuweni](#) (crypto module)
- Java: [Lazysodium for Java](#)
- Java: [jsodium](#)
- Java: [Kalium](#)
- JavaScript (compiled to pure JavaScript): [libsodium.js](#)
- JavaScript (compiled to pure JavaScript): [js-nacl](#)
- JavaScript (libsodium.js wrapper): [Natrium](#)
- JavaScript (libsodium.js wrapper for browsers): [Natrium Browser](#)
- JavaScript (NodeJS): [node-sodium](#)
- JavaScript (NodeJS): [sodium-native](#)
- Julia: [Sodium.jl](#)
- Kotlin/Native: [Sodium](#)
- Lisp (CFFI): [foreign-sodium](#)
- Lua: [lua-sodium](#)
- MRuby: [mruby-libsodium](#)
- MySQL: [MySQL-Sodium](#)
- Nim: [libsodium.nim](#)
- Nim: [Sodium.nim](#)
- OCaml: [ocaml-sodium](#)
- Objective-C: [NAClорide](#)
- Objective-C: [SodiumObjc](#)

- PHP: [PHP-Sodium](#)
- PHP: [libsodium-php](#)
- PHP: [dhole-cryptography](#)
- Perl: [Crypt-Sodium](#)
- Perl: [Crypt::NaCl::Sodium](#)
- Perl: [crypt-sodium-nitrate](#)
- Perl 6: [Natrium](#)
- Pharo/Squeak: [Crypto-NaCl](#)
- Pony: [Pony-Sodium](#)
- Python: [Csodium](#)
- Python: [LibNaCl](#)
- Python: [PyNaCl](#)
- Python: [PySodium](#)
- Q/KDB: [Qsalt](#)
- R: [Cyphr](#)
- R: [Sodium](#)
- Racket: part of [CRESTaceans](#)
- RealBasic and Xojo: [RB-libsodium](#)
- Ruby: [RbNaCl](#)
- Ruby: [Sodium](#)
- Rust: [rust_sodium](#)
- Rust: [Sodium Oxide](#)
- Rust: [libsodium-sys](#)
- Rust: [monosodium](#)
- Scala: [Francium](#)
- Swift: [NaOH](#)
- Swift: [Swift-Sodium](#)
- Swift: [Tafelsalz](#)

Usage

```
#include <sodium.h>

int main(void)
{
    if (sodium_init() == -1) {
        return 1;
    }
    ...
}
```

`sodium.h` is the only header that has to be included.

The library is called `sodium` (use `-lsodium` to link it), and proper compilation/linker flags can be obtained using `pkg-config` on systems where it is available:

```
CFLAGS=$(pkg-config --cflags libsodium)
LDFLAGS=$(pkg-config --libs libsodium)
```

For static linking, Visual Studio users should define `SODIUM_STATIC=1` and `SODIUM_EXPORT=`. This is not required on other platforms.

Projects using CMake can include the [FindSodium.cmake](#) file in order to detect and link the library.

`sodium_init()` initializes the library and should be called before any other function provided by Sodium. It is safe to call this function more than once and from different threads -- subsequent calls won't have any effects.

After this function returns, all of the other functions provided by Sodium will be thread-safe.

`sodium_init()` doesn't perform any memory allocations. However, on Unix systems, it may open `/dev/urandom` and keep the descriptor open, so that the device remains accessible after a `chroot()` call.

Multiple calls to `sodium_init()` do not cause additional descriptors to be opened.

`sodium_init()` returns `0` on success, `-1` on failure, and `1` if the library had already been initialized.

Before returning, the function ensures that the system's random number generator has been properly seeded.

sodium_init() stalling on Linux

On some Linux systems, this may take some time, especially when called right after a reboot of the system. That issue has been reported on Digital Ocean virtual machines as well as on Scaleway ARM instances.

This can be confirmed with the following command:

```
cat /proc/sys/kernel/random/entropy_avail
```

If the command returns `0` or a very low number (`< 160`), and you are not running an obsolete kernel, this is very likely to be the case.

In a virtualized environment, make sure that the `virtio-rng` interface is available. If this is a cloud service and the hypervisor settings are out of your reach, consider switching to a different service.

On a bare-metal host such as Scaleway instances, a possible workaround is to install the `rng-tools` package:

```
apt-get install rng-tools
```

And check the value of `/proc/sys/kernel/random/entropy_avail` again. If the value didn't go any higher, install `haveged` :

```
apt-get install haveged
```

Haveged should only be used as a very last resort. It hasn't received any updates for 10+ years, and shouldn't be trusted as a single entropy source, especially on virtualized environments.

[Jitterentropy](#) is a better alternative, but most Linux distributions don't offer it as an installable package yet.

Applications can warn users about the Linux RNG not being seeded before calling

```
sodium_init()
```

 using code similar to the following:

```
#if defined(__linux__)
# include <fcntl.h>
# include <unistd.h>
# include <sys/ioctl.h>
# include <linux/random.h>
#endif
// ...
#if defined(__linux__) && defined(RNDGETENTCNT)
int fd;
int c;

if ((fd = open("/dev/random", O_RDONLY)) != -1) {
    if (ioctl(fd, RNDGETENTCNT, &c) == 0 && c < 160) {
        fputs("This system doesn't provide enough entropy to quickly generate high-quality random numbers.\n"
              "Installing the rng-utils/rng-tools, jitterentropy or haveged packages may help.\n"
              "On virtualized Linux environments, also consider using virtio-rng.\n"
              "The service will not start until enough entropy has been collected.\n",
              stderr);
    }
    (void) close(fd);
}
#endif
```

Congrats, you're all set up!

A good documentation page to read next might be [Quickstart and FAQ](#).

Helpers

Constant-time test for equality

```
int sodium_memcmp(const void * const b1_, const void * const b2_, size_t len);
```

When a comparison involves secret data (e.g. key, authentication tag), is it critical to use a constant-time comparison function. This property does not relate to computational complexity: it expresses the fact that the time needed to perform the comparison is the same for all data of the same size. The goal is to mitigate side-channel attacks.

The `sodium_memcmp()` function can be used for this purpose.

The function returns `0` if the `len` bytes pointed to by `b1_` match the `len` bytes pointed to by `b2_`. Otherwise, it returns `-1`.

Note: `sodium_memcmp()` is not a lexicographic comparator and is not a generic replacement for `memcmp()`.

Hexadecimal encoding/decoding

```
char *sodium_bin2hex(char * const hex, const size_t hex_maxlen,  
                    const unsigned char * const bin, const size_t bin_len);
```

The `sodium_bin2hex()` function converts `bin_len` bytes stored at `bin` into a hexadecimal string.

The string is stored into `hex` and includes a nul byte (`\0`) terminator.

`hex_maxlen` is the maximum number of bytes that the function is allowed to write starting at `hex`. It must be at least `bin_len * 2 + 1` bytes.

The function always returns `hex`. It evaluates in constant time for a given size.

```
int sodium_hex2bin(unsigned char * const bin, const size_t bin_maxlen,  
                  const char * const hex, const size_t hex_len,  
                  const char * const ignore, size_t * const bin_len,  
                  const char ** const hex_end);
```

The `sodium_hex2bin()` function parses a hexadecimal string `hex` and converts it to a byte sequence.

`hex` does not have to be nul terminated, as the number of characters to parse is supplied via the `hex_len` parameter.

`ignore` is a string of characters to skip. For example, the string `": "` allows columns and spaces to be present at any locations in the hexadecimal string. These characters will just be ignored. As a result, `"69:FC"`, `"69 FC"`, `"69 : FC"` and `"69FC"` will be valid inputs, and will produce the same output.

`ignore` can be set to `NULL` in order to disallow any non-hexadecimal character.

`bin_maxlen` is the maximum number of bytes to put into `bin`.

The parser stops when a non-hexadecimal, non-ignored character is found or when `bin_maxlen` bytes have been written.

If `hex_end` is not `NULL`, it will be set to the address of the first byte after the last valid parsed character.

The function returns `0` on success.

It returns `-1` if more than `bin_maxlen` bytes would be required to store the parsed string, or if the string couldn't be fully parsed, but a valid pointer for `hex_end` was not provided.

It evaluates in constant time for a given length and format.

Base64 encoding/decoding

```
char *sodium_bin2base64(char * const b64, const size_t b64_maxlen,  
                        const unsigned char * const bin, const size_t bin_len,  
                        const int variant);
```

The `sodium_bin2base64()` function encodes `bin` as a Base64 string. `variant` must be one of:

- `sodium_base64_VARIANT_ORIGINAL`
- `sodium_base64_VARIANT_ORIGINAL_NO_PADDING`
- `sodium_base64_VARIANT_URLSAFE`
- `sodium_base64_VARIANT_URLSAFE_NO_PADDING`

None of these Base64 variants provides any form of encryption; just like hex encoding, anyone can decode them.

Computing a correct size for `b64_maxlen` is not straightforward and depends on the chosen variant.

The `sodium_base64_ENCODED_LEN(BIN_LEN, VARIANT)` macro returns the minimum number of bytes required to encode `BIN_LEN` bytes using the Base64 variant `VARIANT`. The returned length includes a trailing `\0` byte.

The `sodium_base64_encoded_len(size_t bin_len, int variant)` function is also available for the same purpose.

```
int sodium_base642bin(unsigned char * const bin, const size_t bin_maxlen,
                      const char * const b64, const size_t b64_len,
                      const char * const ignore, size_t * const bin_len,
                      const char ** const b64_end, const int variant);
```

The `sodium_base642bin()` function decodes a Base64 string using the given variant, and an optional set of characters to ignore (typically: whitespaces and newlines).

If `b64_end` is not `NULL`, it will be set to the address of the first byte after the last valid parsed character.

The function returns `0` on success.

It returns `-1` if more than `bin_maxlen` bytes would be required to store the parsed string, or if the string couldn't be fully parsed, but a valid pointer for `b64_end` was not provided.

Incrementing large numbers

```
void sodium_increment(unsigned char *n, const size_t nlen);
```

The `sodium_increment()` function takes a pointer to an arbitrary-long unsigned number, and increments it.

It runs in constant-time for a given length, and considers the number to be encoded in little-endian format.

`sodium_increment()` can be used to increment nonces in constant time.

Adding large numbers

```
void sodium_add(unsigned char *a, const unsigned char *b, const size_t len);
```


The `sodium_add()` function accepts two pointers to unsigned numbers encoded in little-endian format, `a` and `b`, both of size `len` bytes.

It computes $(a + b) \bmod 2^{(8 \cdot \text{len})}$ in constant time for a given length, and overwrites `a` with the result.

Subtracting large numbers

```
void sodium_sub(unsigned char *a, const unsigned char *b, const size_t len);
```

The `sodium_sub()` function accepts two pointers to unsigned numbers encoded in little-endian format, `a` and `b`, both of size `len` bytes.

It computes $(a - b) \bmod 2^{(8 \cdot \text{len})}$ in constant time for a given length, and overwrites `a` with the result.

This function was introduced in libsodium 1.0.17.

Comparing large numbers

```
int sodium_compare(const void * const b1_, const void * const b2_, size_t len);
```

Given `b1_` and `b2_`, two `len` bytes numbers encoded in little-endian format, this function returns:

- `-1` if `b1_` is less than `b2_`
- `0` if `b1_` equals `b2_`
- `1` if `b1_` is greater than `b2_`

The comparison is done in constant time for a given length.

This function can be used with nonces, in order to prevent replay attacks.

Testing for all zeros

```
int sodium_is_zero(const unsigned char *n, const size_t nlen);
```

This function returns `1` if the `nlen` bytes vector pointed by `n` contains only zeros. It returns `0` if non-zero bits are found.

Its execution time is constant for a given length.

Clearing the stack

```
void sodium_stackzero(const size_t len);
```

The `sodium_stackzero()` function clears `len` bytes above the current stack pointer, to overwrite sensitive values that may have been temporarily stored on the stack.

Note that these values can still be present in registers.

This function was introduced in libsodium 1.0.16.

Notes

The `sodium_base64_VARIANT_*` macros don't have associated symbols. Bindings are encouraged to define specialized encoding/decoding functions instead.

Padding

Most modern cryptographic constructions disclose message lengths. The ciphertext for a given message will always have the same length, or add a constant number of bytes to it.

For most applications, this is not an issue. But in some specific situations, such as interactive remote shells, hiding the length may be desirable. Padding can be used for that purpose.

This API was introduced in libsodium 1.0.14.

Example

```
unsigned char buf[100];
size_t      buf_unpadded_len = 10;
size_t      buf_padded_len;
size_t      block_size = 16;

/* round the length of the buffer to a multiple of `block_size` by appending
 * padding data and put the new, total length into `buf_padded_len` */
if (sodium_pad(&buf_padded_len, buf, buf_unpadded_len, block_size, sizeof buf) != 0) {
    /* overflow! buf[] is not large enough */
}

/* compute the original, unpadded length */
if (sodium_unpad(&buf_unpadded_len, buf, buf_padded_len, block_size) != 0) {
    /* incorrect padding */
}
```

Usage

```
int sodium_pad(size_t *padded_buflen_p, unsigned char *buf,
               size_t unpadded_buflen, size_t blocksize, size_t max_buflen);
```

The `sodium_pad()` function adds padding data to a buffer `buf` whose original size is `unpadded_buflen` in order to extend its total length to a multiple of `blocksize`.

The new length is put into `padded_buflen_p`.

The function returns `-1` if the padded buffer length would exceed `max_buflen`, or if the block size is `0`. It returns `0` on success.

```
int sodium_unpad(size_t *unpadded_buflen_p, const unsigned char *buf,
                 size_t padded_buflen, size_t blocksize);
```

The `sodium_unpad()` function computes the original, unpadded length of a message previously padded using `sodium_pad()`. The original length is put into `unpadded_buflen_p`.

Algorithm

These functions use the ISO/IEC 7816-4 padding algorithm. It supports arbitrary block sizes, ensures that the padding data are checked for computing the unpadded length, and is more resistant to some classes of attacks than other standard padding algorithms.

Notes

Padding should be applied prior to encryption, and removed after decryption.

Usage of padding in order to hide the length of a password is not recommended. A client willing to send a password to a server should hash it instead (even with a single iteration of the hash function).

This ensures that the length of the transmitted data is constant, and that the server doesn't effortlessly get a copy of the password.

Applications may eventually leak the unpadded length via side channels, but the `sodium_pad()` and `sodium_unpad()` functions themselves try to minimize side channels for a given `length & <block size mask>` value.

Securing memory allocations

Zeroing memory

```
void sodium_memzero(void * const pnt, const size_t len);
```

After use, sensitive data should be overwritten, but `memset()` and hand-written code can be silently stripped out by an optimizing compiler or by the linker.

The `sodium_memzero()` function tries to effectively zero `len` bytes starting at `pnt`, even if optimizations are being applied to the code.

Locking memory

```
int sodium_mlock(void * const addr, const size_t len);
```

The `sodium_mlock()` function locks at least `len` bytes of memory starting at `addr`. This can help avoid swapping sensitive data to disk.

In addition, it is recommended to totally disable swap partitions on machines processing sensitive data, or, as a second choice, use encrypted swap partitions.

For similar reasons, on Unix systems, one should also disable core dumps when running crypto code outside a development environment. This can be achieved using a shell built-in such as `ulimit` or programatically using `setrlimit(RLIMIT_CORE, &(struct rlimit) {0, 0})`. On operating systems where this feature is implemented, kernel crash dumps should also be disabled.

`sodium_mlock()` wraps `mlock()` and `VirtualLock()`. **Note:** Many systems place limits on the amount of memory that may be locked by a process. Care should be taken to raise those limits (e.g. Unix ulimits) where necessary. `sodium_mlock()` will return `-1` when any limit is reached.

```
int sodium_munlock(void * const addr, const size_t len);
```

The `sodium_munlock()` function should be called after locked memory is not being used any more. It will zero `len` bytes starting at `addr` before actually flagging the pages as swappable again. Calling `sodium_memzero()` prior to `sodium_munlock()` is thus not required.

On systems where it is supported, `sodium_mlock()` also wraps `madvise()` and advises the kernel not to include the locked memory in core dumps. `sodium_munlock()` also undoes this additional protection.

Guarded heap allocations

Sodium provides heap allocation functions for storing sensitive data.

These are not general-purpose allocation functions. In particular, they are slower than `malloc()` and friends, and they require 3 or 4 extra pages of virtual memory.

`sodium_init()` has to be called before using any of the guarded heap allocation functions.

```
void *sodium_malloc(size_t size);
```

The `sodium_malloc()` function returns a pointer from which exactly `size` contiguous bytes of memory can be accessed. Like normal `malloc`, `NULL` may be returned and `errno` set if it is not possible to allocate enough memory.

The allocated region is placed at the end of a page boundary, immediately followed by a guard page. As a result, accessing memory past the end of the region will immediately terminate the application.

A canary is also placed right before the returned pointer. Modifications of this canary are detected when trying to free the allocated region with `sodium_free()`, and also cause the application to immediately terminate.

An additional guard page is placed before this canary to make it less likely for sensitive data to be accessible when reading past the end of an unrelated region.

The allocated region is filled with `0xdb` bytes in order to help catch bugs due to uninitialized data.

In addition, `sodium_mlock()` is called on the region to help avoid it being swapped to disk. On operating systems supporting `MAP_NOCORE` or `MADV_DONTDUMP`, memory allocated this way will also not be part of core dumps.

The returned address will not be aligned if the allocation size is not a multiple of the required alignment.

For this reason, `sodium_malloc()` should not be used with packed or variable-length structures, unless the size given to `sodium_malloc()` is rounded up in order to ensure proper alignment.

All the structures used by libsodium can safely be allocated using `sodium_malloc()`.

Allocating 0 bytes is a valid operation. It returns a pointer that can be successfully passed to `sodium_free()`.

```
void *sodium_allocarray(size_t count, size_t size);
```

The `sodium_allocarray()` function returns a pointer from which `count` objects that are `size` bytes of memory each can be accessed.

It provides the same guarantees as `sodium_malloc()` but also protects against arithmetic overflows when `count * size` exceeds `SIZE_MAX`.

```
void sodium_free(void *ptr);
```

The `sodium_free()` function unlocks and deallocates memory allocated using `sodium_malloc()` or `sodium_allocarray()`.

Prior to this, the canary is checked in order to detect possible buffer underflows and terminate the process if required.

`sodium_free()` also fills the memory region with zeros before the deallocation.

This function can be called even if the region was previously protected using `sodium_mprotect_readonly()`; the protection will automatically be changed as needed.

`ptr` can be `NULL`, in which case no operation is performed.

```
int sodium_mprotect_noaccess(void *ptr);
```

The `sodium_mprotect_noaccess()` function makes a region allocated using `sodium_malloc()` or `sodium_allocarray()` inaccessible. It cannot be read or written, but the data are preserved.

This function can be used to make confidential data inaccessible except when actually needed for a specific operation.

```
int sodium_mprotect_readonly(void *ptr);
```

The `sodium_mprotect_readonly()` function marks a region allocated using `sodium_malloc()` or `sodium_allocarray()` as read-only.

Attempting to modify the data will cause the process to terminate.

```
int sodium_mprotect_readwrite(void *ptr);
```

The `sodium_mprotect_readwrite()` function marks a region allocated using `sodium_malloc()` or `sodium_allocarray()` as readable and writable, after having been protected using `sodium_mprotect_readonly()` or `sodium_mprotect_noaccess()` .

Generating random data

The library provides a set of functions to generate unpredictable data, suitable for creating secret keys.

- On Windows systems, the `RtlGenRandom()` function is used
- On OpenBSD and Bitrig, the `arc4random()` function is used
- On recent FreeBSD and Linux kernels, the `getrandom` system call is used
- On other Unices, the `/dev/urandom` device is used
- If none of these options can safely be used, custom implementations can easily be hooked.

Usage

```
uint32_t randombytes_random(void);
```

The `randombytes_random()` function returns an unpredictable value between `0` and `0xffffffff` (included).

```
uint32_t randombytes_uniform(const uint32_t upper_bound);
```

The `randombytes_uniform()` function returns an unpredictable value between `0` and `upper_bound` (excluded). Unlike `randombytes_random() % upper_bound`, it guarantees a uniform distribution of the possible output values even when `upper_bound` is not a power of 2. Note that an `upper_bound < 2` leaves only a single element to be chosen, namely `0`.

```
void randombytes_buf(void * const buf, const size_t size);
```

The `randombytes_buf()` function fills `size` bytes starting at `buf` with an unpredictable sequence of bytes.

```
void randombytes_buf_deterministic(void * const buf, const size_t size,  
                                   const unsigned char seed[randombytes_SEEDBYTES]);
```

The `randombytes_buf_deterministic` function stores `size` bytes into `buf` indistinguishable from random bytes without knowing `seed`.

For a given `seed`, this function will always output the same sequence. `size` can be up to 2^{38} (256 GB).

`seed` is `randombytes_SEEDBYTES` bytes long.

This function is mainly useful for writing tests, and was introduced in libsodium 1.0.12. Under the hood, it uses the ChaCha20 stream cipher.

Up to 256 GB can be produced with a single seed.

```
int randombytes_close(void);
```

This deallocates the global resources used by the pseudo-random number generator. More specifically, when the `/dev/urandom` device is used, it closes the descriptor. Explicitly calling this function is almost never required.

```
void randombytes_stir(void);
```

The `randombytes_stir()` function reseeds the pseudo-random number generator, if it supports this operation. Calling this function is not required with the default generator, even after a `fork()` call, unless the descriptor for `/dev/urandom` was closed using `randombytes_close()`.

If a non-default implementation is being used (see `randombytes_set_implementation()`), `randombytes_stir()` must be called by the child after a `fork()` call.

Note

If this is used in an application inside a VM, and the VM is snapshotted and restored, then the above functions may produce the same output.

Secret-key cryptography

Secret-key cryptography refers to cryptographic system that uses the **same key** to encrypt and decrypt data.

This means that all parties involved have to know the key to be able to communicate securely – that is, decrypt encrypted messages to read them and encrypt messages they want to send.

Therefore the key, being *shared* among parties, but having to stay *secret* to 3rd parties – in order to keep communications private – is considered a *shared secret*.

Using secret-key cryptography, Alice and Bob would have to devise a **single** cryptographic key that they will **both** know and use each time they send each other a message.

Alice encrypts her message using this shared key, sends the *ciphertext* (the message, once encrypted) to Bob, then Bob uses the same key again to decrypt, ultimately reading the message.

An adversary cannot decrypt the message without knowing the secret key.

Such cryptographic system, requiring the key to be shared by both parties, is also known as *symmetric-key cryptography*.

Secret-key authenticated encryption

Example

```
#define MESSAGE ((const unsigned char *) "test")
#define MESSAGE_LEN 4
#define CIPHERTEXT_LEN (crypto_secretbox_MACBYTES + MESSAGE_LEN)

unsigned char key[crypto_secretbox_KEYBYTES];
unsigned char nonce[crypto_secretbox_NONCEBYTES];
unsigned char ciphertext[CIPHERTEXT_LEN];

crypto_secretbox_keygen(key);
randombytes_buf(nonce, sizeof nonce);
crypto_secretbox_easy(ciphertext, MESSAGE, MESSAGE_LEN, nonce, key);

unsigned char decrypted[MESSAGE_LEN];
if (crypto_secretbox_open_easy(decrypted, ciphertext, CIPHERTEXT_LEN, nonce, key) != 0
) {
    /* message forged! */
}
```

Purpose

This operation:

- Encrypts a message with a key and a nonce to keep it confidential
- Computes an authentication tag. This tag is used to make sure that the message hasn't been tampered with before decrypting it.

A single key is used both to encrypt/authenticate and verify/decrypt messages. For this reason, it is critical to keep the key confidential.

The nonce doesn't have to be confidential, but it should never ever be reused with the same key. The easiest way to generate a nonce is to use `randombytes_buf()`.

Messages encrypted are assumed to be independent. If multiple messages are sent using this API and random nonces, there will be no way to detect if a message has been received twice, or if messages have been reordered. If this is a requirement, see the [encrypting a sequence of a set of related messages](#) section.

Combined mode

In combined mode, the authentication tag and the encrypted message are stored together. This is usually what you want.

```
int crypto_secretbox_easy(unsigned char *c, const unsigned char *m,
                          unsigned long long mlen, const unsigned char *n,
                          const unsigned char *k);
```

The `crypto_secretbox_easy()` function encrypts a message `m` whose length is `mlen` bytes, with a key `k` and a nonce `n`.

`k` should be `crypto_secretbox_KEYBYTES` bytes and `n` should be `crypto_secretbox_NONCEBYTES` bytes.

`c` should be at least `crypto_secretbox_MACBYTES + mlen` bytes long.

This function writes the authentication tag, whose length is `crypto_secretbox_MACBYTES` bytes, in `c`, immediately followed by the encrypted message, whose length is the same as the plaintext: `mlen`.

`c` and `m` can overlap, making in-place encryption possible. However do not forget that `crypto_secretbox_MACBYTES` extra bytes are required to prepend the tag.

```
int crypto_secretbox_open_easy(unsigned char *m, const unsigned char *c,
                              unsigned long long clen, const unsigned char *n,
                              const unsigned char *k);
```

The `crypto_secretbox_open_easy()` function verifies and decrypts a ciphertext produced by `crypto_secretbox_easy()`.

`c` is a pointer to an authentication tag + encrypted message combination, as produced by `crypto_secretbox_easy()`. `clen` is the length of this authentication tag + encrypted message combination. Put differently, `clen` is the number of bytes written by `crypto_secretbox_easy()`, which is `crypto_secretbox_MACBYTES` + the length of the message.

The nonce `n` and the key `k` have to match the used to encrypt and authenticate the message.

The function returns `-1` if the verification fails, and `0` on success. On success, the decrypted message is stored into `m`.

`m` and `c` can overlap, making in-place decryption possible.

Detached mode

Some applications may need to store the authentication tag and the encrypted message at different locations.

For this specific use case, "detached" variants of the functions above are available.

```
int crypto_secretbox_detached(unsigned char *c, unsigned char *mac,
                              const unsigned char *m,
                              unsigned long long mlen,
                              const unsigned char *n,
                              const unsigned char *k);
```

This function encrypts a message `m` of length `mlen` with a key `k` and a nonce `n`, and puts the encrypted message into `c`. Exactly `mlen` bytes will be put into `c`, since this function does not prepend the authentication tag. The tag, whose size is

`crypto_secretbox_MACBYTES` bytes, will be put into `mac`.

```
int crypto_secretbox_open_detached(unsigned char *m,
                                    const unsigned char *c,
                                    const unsigned char *mac,
                                    unsigned long long clen,
                                    const unsigned char *n,
                                    const unsigned char *k);
```

The `crypto_secretbox_open_detached()` function verifies and decrypts an encrypted message `c` whose length is `clen`. `clen` doesn't include the tag, so this length is the same as the plaintext.

The plaintext is put into `m` after verifying that `mac` is a valid authentication tag for this ciphertext, with the given nonce `n` and key `k`.

The function returns `-1` if the verification fails, or `0` on success.

```
void crypto_secretbox_keygen(unsigned char k[crypto_secretbox_KEYBYTES]);
```

This helper function introduced in libsodium 1.0.12 creates a random key `k`.

It is equivalent to calling `randombytes_buf()` but improves code clarity and can prevent misuse by ensuring that the provided key length is always be correct.

Constants

- `crypto_secretbox_KEYBYTES`
- `crypto_secretbox_MACBYTES`
- `crypto_secretbox_NONCEBYTES`

Algorithm details

- Encryption: XSalsa20 stream cipher
- Authentication: Poly1305 MAC

Notes

Internally, `crypto_secretbox` calls `crypto_stream_xor()` to encrypt the message. As a result, a secret key used with the former should not be reused with the later. But as a general rule, a key should not be reused for different purposes.

The original NaCl `crypto_secretbox` API is also supported, albeit not recommended.

`crypto_secretbox()` takes a pointer to 32 bytes before the message, and stores the ciphertext 16 bytes after the destination pointer, the first 16 bytes being overwritten with zeros. `crypto_secretbox_open()` takes a pointer to 16 bytes before the ciphertext and stores the message 32 bytes after the destination pointer, overwriting the first 32 bytes with zeros.

The `_easy` and `_detached` APIs are faster and improve usability by not requiring padding, copying or tricky pointer arithmetic.

Stream encryption/file encryption

This high-level API encrypts a sequence of messages, or a single message split into an arbitrary number of chunks, using a secret key, with the following properties:

- Messages cannot be truncated, removed, reordered, duplicated or modified without this being detected by the decryption functions.
- The same sequence encrypted twice will produce different ciphertexts.
- An authentication tag is added to each encrypted message: stream corruption will be detected early, without having to read the stream until the end.
- Each message can include additional data (ex: timestamp, protocol version) in the computation of the authentication tag.
- Messages can have different sizes.
- There are no practical limits to the total length of the stream, or to the total number of individual messages.
- Ratcheting: at any point in the stream, it is possible to "forget" the key used to encrypt the previous messages, and switch to a new key.

This API can be used to securely send an ordered sequence of messages to a peer. Since the length of the stream is not limited, it can also be used to encrypt files regardless of their size.

It transparently generates nonces and automatically handles key rotation.

The `crypto_secretstream_*` API was introduced in libsodium 1.0.14.

Example (stream encryption)


```

#define MESSAGE_PART1 (const unsigned char *) "Arbitrary data to encrypt"
#define MESSAGE_PART1_LEN    25
#define CIPHERTEXT_PART1_LEN MESSAGE_PART1_LEN + crypto_secretstream_xchacha20poly1305_
_ABYTES

#define MESSAGE_PART2 (const unsigned char *) "split into"
#define MESSAGE_PART2_LEN    10
#define CIPHERTEXT_PART2_LEN MESSAGE_PART2_LEN + crypto_secretstream_xchacha20poly1305_
_ABYTES

#define MESSAGE_PART3 (const unsigned char *) "three messages"
#define MESSAGE_PART3_LEN    14
#define CIPHERTEXT_PART3_LEN MESSAGE_PART3_LEN + crypto_secretstream_xchacha20poly1305_
_ABYTES

crypto_secretstream_xchacha20poly1305_state state;
unsigned char key[crypto_secretstream_xchacha20poly1305_KEYBYTES];
unsigned char header[crypto_secretstream_xchacha20poly1305_HEADERBYTES];
unsigned char c1[CIPHERTEXT_PART1_LEN],
              c2[CIPHERTEXT_PART2_LEN],
              c3[CIPHERTEXT_PART3_LEN];

/* Shared secret key required to encrypt/decrypt the stream */
crypto_secretstream_xchacha20poly1305_keygen(key);

/* Set up a new stream: initialize the state and create the header */
crypto_secretstream_xchacha20poly1305_init_push(&state, header, key);

/* Now, encrypt the first chunk. `c1` will contain an encrypted,
 * authenticated representation of `MESSAGE_PART1`. */
crypto_secretstream_xchacha20poly1305_push
(&state, c1, NULL, MESSAGE_PART1, MESSAGE_PART1_LEN, NULL, 0, 0);

/* Encrypt the second chunk. `c2` will contain an encrypted, authenticated
 * representation of `MESSAGE_PART2`. */
crypto_secretstream_xchacha20poly1305_push
(&state, c2, NULL, MESSAGE_PART2, MESSAGE_PART2_LEN, NULL, 0, 0);

/* Encrypt the last chunk, and store the ciphertext into `c3`.
 * Note the `TAG_FINAL` tag to indicate that this is the final chunk. */
crypto_secretstream_xchacha20poly1305_push
(&state, c3, NULL, MESSAGE_PART3, MESSAGE_PART3_LEN, NULL, 0,
 crypto_secretstream_xchacha20poly1305_TAG_FINAL);

```

Example (stream decryption)

```

unsigned char tag;
unsigned char m1[MESSAGE_PART1_LEN],
             m2[MESSAGE_PART2_LEN],
             m3[MESSAGE_PART3_LEN];

/* Decrypt the stream: initializes the state, using the key and a header */
if (crypto_secretstream_xchacha20poly1305_init_pull(&state, header, key) != 0) {
    /* Invalid header, no need to go any further */
}

/* Decrypt the first chunk. A real application would probably use
 * a loop, that reads data from the network or from disk, and exits after
 * an error, or after the last chunk (with a `TAG_FINAL` tag) has been
 * decrypted. */
if (crypto_secretstream_xchacha20poly1305_pull
    (&state, m1, NULL, &tag, c1, CIPHERTEXT_PART1_LEN, NULL, 0) != 0) {
    /* Invalid/incomplete/corrupted ciphertext - abort */
}
assert(tag == 0); /* The tag is the one we attached to this chunk: 0 */

/* Decrypt the second chunk, store the result into `m2` */
if (crypto_secretstream_xchacha20poly1305_pull
    (&state, m2, NULL, &tag, c2, CIPHERTEXT_PART2_LEN, NULL, 0) != 0) {
    /* Invalid/incomplete/corrupted ciphertext - abort */
}
assert(tag == 0); /* Not the end of the stream yet */

/* Decrypt the last chunk, store the result into `m3` */
if (crypto_secretstream_xchacha20poly1305_pull
    (&state, m3, NULL, &tag, c3, CIPHERTEXT_PART3_LEN, NULL, 0) != 0) {
    /* Invalid/incomplete/corrupted ciphertext - abort */
}
/* The tag indicates that this is the final chunk, no need to read and decrypt more */
assert(tag == crypto_secretstream_xchacha20poly1305_TAG_FINAL);

```

See down below for a complete example of file encryption/decryption using the secretstream API.

Usage

The `crypto_secretstream_*_push()` functions set creates an encrypted stream. The `crypto_secretstream_*_pull()` functions set is the decryption counterpart.

An encrypted stream starts with a short header, whose size is

`crypto_secretstream_xchacha20poly1305_HEADERBYTES` bytes. That header must be sent/stored before the sequence of encrypted messages, as it is required to decrypt the stream.

A tag is attached to each message. That tag can be any of:

- `0` , Or `crypto_secretstream_xchacha20poly1305_TAG_MESSAGE` : the most common tag, that doesn't add any information about the nature of the message.
- `crypto_secretstream_xchacha20poly1305_TAG_FINAL` : indicates that the message marks the end of the stream, and erases the secret key used to encrypt the previous sequence.
- `crypto_secretstream_xchacha20poly1305_TAG_PUSH` : indicates that the message marks the end of a set of messages, but not the end of the stream. For example, a huge JSON string sent as multiple chunks can use this tag to indicate to the application that the string is complete and that it can be decoded. But the stream itself is not closed, and more data may follow.
- `crypto_secretstream_xchacha20poly1305_TAG_REKEY` : "forget" the key used to encrypt this message and the previous ones, and derive a new secret key.

A typical encrypted stream simply attaches `0` as a tag to all messages, except the last one which is tagged as `TAG_FINAL` .

Note that tags are encrypted; encrypted streams do not reveal any information about sequence boundaries (`PUSH` and `REKEY` tags).

For each message, additional data can be included in the computation of the authentication tag. With this API, additional data is rarely required, and most applications can just use `NULL` and a length of `0` instead.

Encryption

```
void crypto_secretstream_xchacha20poly1305_keygen
(unsigned char k[crypto_secretstream_xchacha20poly1305_KEYBYTES]);
```

Creates a random, secret key to encrypt a stream, and stores it into `k` .

Note that using this function is not required to obtain a suitable key: the `secretstream` API can use any secret key whose size is `crypto_secretstream_xchacha20poly1305_KEYBYTES` bytes.

Network protocols can leverage the key exchange API in order to get a shared key that can be used to encrypt streams. Similarly, file encryption applications can use the password hashing API to get a key that can be used with the functions below.

```
int crypto_secretstream_xchacha20poly1305_init_push
(crypto_secretstream_xchacha20poly1305_state *state,
 unsigned char header[crypto_secretstream_xchacha20poly1305_HEADERBYTES],
 const unsigned char k[crypto_secretstream_xchacha20poly1305_KEYBYTES]);
```

The `crypto_secretstream_xchacha20poly1305_init_push()` function initializes a state `state` using the key `k` and an internal, automatically generated initialization vector. It then stores the stream header into `header` (`crypto_secretstream_xchacha20poly1305_HEADERBYTES` bytes).

This is the first function to call in order to create an encrypted stream. The key `k` will not be required any more for subsequent operations.

```
int crypto_secretstream_xchacha20poly1305_push
(crypto_secretstream_xchacha20poly1305_state *state,
 unsigned char *c, unsigned long long *clen_p,
 const unsigned char *m, unsigned long long mlen,
 const unsigned char *ad, unsigned long long adlen, unsigned char tag);
```

The `crypto_secretstream_xchacha20poly1305_push()` function encrypts a message `m` of length `mlen` bytes using the state `state` and the tag `tag`.

Additional data `ad` of length `adlen` can be included in the computation of the authentication tag. If no additional data is required, `ad` can be `NULL` and `adlen` set to `0`.

The ciphertext is put into `c`.

If `clen_p` is not `NULL`, the ciphertext length will be stored at that address. But with this particular construction, the ciphertext length is guaranteed to always be `mlen + crypto_secretstream_xchacha20poly1305_ABYTES`.

The maximum length of an individual message is

`crypto_secretstream_xchacha20poly1305_MESSAGEBYTES_MAX` bytes (~ 256 GB).

Decryption

```
int crypto_secretstream_xchacha20poly1305_init_pull
(crypto_secretstream_xchacha20poly1305_state *state,
 const unsigned char header[crypto_secretstream_xchacha20poly1305_HEADERBYTES],
 const unsigned char k[crypto_secretstream_xchacha20poly1305_KEYBYTES]);
```

The `crypto_secretstream_xchacha20poly1305_init_pull()` function initializes a state given a secret key `k` and a header `header`. The key `k` will not be required any more for subsequent operations.

It returns `0` on success, or `-1` if the header is invalid.

```
int crypto_secretstream_xchacha20poly1305_pull
(crypto_secretstream_xchacha20poly1305_state *state,
 unsigned char *m, unsigned long long *mlen_p, unsigned char *tag_p,
 const unsigned char *c, unsigned long long clen,
 const unsigned char *ad, unsigned long long adlen);
```

The `crypto_secretstream_xchacha20poly1305_pull()` function verifies that `c` (a sequence of `clen` bytes) contains a valid ciphertext and authentication tag for the given state `state` and optional authenticated data `ad` of length `adlen` bytes.

If the ciphertext appears to be invalid, the function returns `-1`.

If the authentication tag appears to be correct, the decrypted message is put into `m`.

If `tag_p` is not `NULL`, the tag attached to the message is stored at that address.

If `mlen_p` is not `NULL`, the message length is stored at that address. But with this particular construction, it is guaranteed to always be `clen - crypto_secretstream_xchacha20poly1305_ABYTES` bytes.

Applications will typically call this function in a loop, until a message with the `crypto_secretstream_xchacha20poly1305_TAG_FINAL` tag is found.

Rekeying

Rekeying happens automatically and transparently, before the internal counter of the underlying cipher wraps. Therefore, streams can be arbitrary large.

Optionally, applications for which forward secrecy is critical can attach the `crypto_secretstream_xchacha20poly1305_TAG_REKEY` tag to a message in order to trigger an explicit rekeying. The decryption API will automatically update the key if this tag is found attached to a message.

Explicit rekeying can also be performed without adding a tag, by calling the `crypto_secretstream_xchacha20poly1305_rekey()` function:

```
void crypto_secretstream_xchacha20poly1305_rekey
(crypto_secretstream_xchacha20poly1305_state *state);
```

This updates the state, but doesn't add any information about the key change to the stream. If this function is used to create an encrypted stream, the decryption process must call that function at the exact same stream location.

Constants

- `crypto_secretstream_xchacha20poly1305_ABYTES`
- `crypto_secretstream_xchacha20poly1305_HEADERBYTES`
- `crypto_secretstream_xchacha20poly1305_KEYBYTES`
- `crypto_secretstream_xchacha20poly1305_MESSAGEBYTES_MAX`
- `crypto_secretstream_xchacha20poly1305_TAG_MESSAGE`
- `crypto_secretstream_xchacha20poly1305_TAG_PUSH`
- `crypto_secretstream_xchacha20poly1305_TAG_REKEY`
- `crypto_secretstream_xchacha20poly1305_TAG_FINAL`

Algorithm

Initialization (`secretstream_init`): a subkey `k` and a 64-bit nonce `n` are derived from a key `κ` and a 192-bit random nonce `N`, using the same algorithm as XChaCha20. `i` is a 32-bit counter.

```
k <- HChaCha20(K, n[0..16])
n <- N[16..24]
i <- 1
```

`secretstream_init_push()` outputs `N`.

Encryption:

For every message `M` with a tag `T`:

```
c, mac <- ChaCha20Poly1305-IETF-XOR(key = k, nonce = i || n, msg = T || {0} * 63 || M)
n <- n ^ mac
i <- (i + 1) & 0xffffffff
if i = 0:
  rekey()
```

`secretstream_push()` outputs `c` with the first block truncated to the tag size: `c[0] || c[64..] || mac`

Encrypting a unique message using `secretstream` is equivalent to `XChaCha20Poly1305-IETF-XOR(key = k, nonce = 1 || n, T || {0} * 63 || M)`.

Rekeying:

```
k || n <- ChaCha20-IETF-XOR(key = k, nonce = i || n, msg = k || n)
i <- 1
```

A `FINAL` tag performs an implicit rekeying.

File encryption example code

```
#include <stdio.h>
#include <sodium.h>

#define CHUNK_SIZE 4096

static int
encrypt(const char *target_file, const char *source_file,
        const unsigned char key[crypto_secretstream_xchacha20poly1305_KEYBYTES])
{
    unsigned char buf_in[CHUNK_SIZE];
    unsigned char buf_out[CHUNK_SIZE + crypto_secretstream_xchacha20poly1305_ABYTES];
    unsigned char header[crypto_secretstream_xchacha20poly1305_HEADERBYTES];
    crypto_secretstream_xchacha20poly1305_state st;
    FILE *fp_t, *fp_s;
    unsigned long long out_len;
    size_t rlen;
    int eof;
    unsigned char tag;

    fp_s = fopen(source_file, "rb");
    fp_t = fopen(target_file, "wb");
    crypto_secretstream_xchacha20poly1305_init_push(&st, header, key);
    fwrite(header, 1, sizeof header, fp_t);
    do {
        rlen = fread(buf_in, 1, sizeof buf_in, fp_s);
        eof = feof(fp_s);
        tag = eof ? crypto_secretstream_xchacha20poly1305_TAG_FINAL : 0;
        crypto_secretstream_xchacha20poly1305_push(&st, buf_out, &out_len, buf_in, rlen,
n,
                                                    NULL, 0, tag);
        fwrite(buf_out, 1, (size_t) out_len, fp_t);
    } while (! eof);
    fclose(fp_t);
    fclose(fp_s);
    return 0;
}

static int
decrypt(const char *target_file, const char *source_file,
        const unsigned char key[crypto_secretstream_xchacha20poly1305_KEYBYTES])
{
    unsigned char buf_in[CHUNK_SIZE + crypto_secretstream_xchacha20poly1305_ABYTES];
```

```

unsigned char  buf_out[CHUNK_SIZE];
unsigned char  header[crypto_secretstream_xchacha20poly1305_HEADERBYTES];
crypto_secretstream_xchacha20poly1305_state st;
FILE           *fp_t, *fp_s;
unsigned long long out_len;
size_t         rlen;
int            eof;
int            ret = -1;
unsigned char  tag;

fp_s = fopen(source_file, "rb");
fp_t = fopen(target_file, "wb");
fread(header, 1, sizeof header, fp_s);
if (crypto_secretstream_xchacha20poly1305_init_pull(&st, header, key) != 0) {
    goto ret; /* incomplete header */
}
do {
    rlen = fread(buf_in, 1, sizeof buf_in, fp_s);
    eof = feof(fp_s);
    if (crypto_secretstream_xchacha20poly1305_pull(&st, buf_out, &out_len, &tag,
                                                    buf_in, rlen, NULL, 0) != 0) {
        goto ret; /* corrupted chunk */
    }
    if (tag == crypto_secretstream_xchacha20poly1305_TAG_FINAL && ! eof) {
        goto ret; /* premature end (end of file reached before the end of the stream) */
    }
    fwrite(buf_out, 1, (size_t) out_len, fp_t);
} while (! eof);

ret = 0;
ret:
fclose(fp_t);
fclose(fp_s);
return ret;
}

int
main(void)
{
    unsigned char key[crypto_secretstream_xchacha20poly1305_KEYBYTES];

    if (sodium_init() != 0) {
        return 1;
    }
    crypto_secretstream_xchacha20poly1305_keygen(key);
    if (encrypt("/tmp/encrypted", "/tmp/original", key) != 0) {
        return 1;
    }
    if (decrypt("/tmp/decrypted", "/tmp/encrypted", key) != 0) {
        return 1;
    }
    return 0;
}

```



```
}
```

Encrypting a sequence or a set of dependent messages

The `crypto_secretbox`, `crypto_box` and `crypto_seal` APIs are designed to encrypt **independent** messages.

However, applications may wish to encrypt a set of messages with the following constraints:

- If a sequence of messages is encrypted, the decryption system must ensure that the complete, unmodified sequence has been properly received. In particular, it must guarantee that messages haven't been added, removed, duplicated, truncated or reordered.
- If an unordered set of encrypted messages is transmitted (for example when using a protocol such as UDP), the decryption system must be able to reorder the messages.

Simply encrypting individual messages with a random nonce doesn't respect these constraints.

For sequences of messages, libsodium 1.0.14 and beyond implement the `crypto_secretstream` API that satisfies the above constraints. This API is recommended to encrypt files or for secure communications over a reliable protocol with ordering guarantees, such as TCP.

On older versions of the library, and with transport protocols featuring weaker ordering and reliability guarantees, these constraints can be satisfied using AEAD (Authenticated Encryption with Additional Data) constructions.

Recommended API: `crypto_aead_xchacha20poly1305_ietf_*` .

Initialization vector

The same key will be used to encrypt a set of messages. This is perfectly acceptable as long as that key is combined with a unique nonce for each message.

The easiest way to achieve this is to choose a random initial nonce, and to increment it after each message:

```
unsigned char nonce[NPUBBYTES];

randombytes_buf(nonce, sizeof nonce);

encrypt(c1, m1, key, nonce);
sodium_increment(nonce, sizeof nonce);
encrypt(c2, m2, key, nonce);
sodium_increment(nonce, sizeof nonce);
...
```

Although the terms "nonce" and "initialization vector" are frequently used interchangeably, in this documentation, we use the term "initialization vector" to describe the first nonce used to encrypt a set of messages.

A random initialization vector ensures that even if the key is reused to encrypt a different set of messages, two messages will not be encrypted using the same nonce, which, for most constructions, is critical for security.

This assumes that the size of the nonce is big enough that the probability of a nonce reuse is negligible. With a 192-bit nonce size, the XChaCha20 and XSalsa20 ciphers fit in this category. The AES and ChaCha20 (not XChaCha20) ciphers do *not* fit in this category. Please refer to the "short nonces" section below for recommendations about using such ciphers, if you really have to use these.

In order for the set of messages to be decrypted, the initialization vector must be transmitted. Unlike the key, it doesn't have to be secret, so it can be prepended to the ciphertext, before the first message.

Authentication tags

AEAD constructions encrypt messages and append an authentication tag. That authentication tag is computed using the following data:

- The secret key
- The nonce
- The message, before or after encryption
- Optionally, some additional data

During the decryption process, the authentication tag is computed using the same data, and compared with the one that was attached to the ciphertext. If the tag is large enough, a valid authentication tag for a given ciphertext cannot be computed without knowing the secret key.

If these authentication tags differ, it means that the ciphertext has been corrupted, tampered with, or created without the correct secret key. In such a situation, decryption functions return an error code, and applications must discard the invalid received data.

Note that the nonce is included in the computation of the tag. A valid tag for a given ciphertext and nonce will not verify with the same ciphertext, but a different nonce.

Additional data

As described above, the computation of an authentication tag can include additional data. This is completely optional, and most applications don't need to include any.

For a given key, nonce and message, the authentication tag will be different if the additional data differs. The ciphertext will be the same, though. Additional data are usually non-secret data.

How additional data are used is specific to every application and protocol, but here are two sample use cases for them:

- **version identifiers:** using a version identifier as additional data allows the recipient of an encrypted message to check that the expected protocol version was used. If a valid secret key is used, but the version is not the one expected by the recipient, decryption will fail.
- **timestamps:** when using a datagram-based transport protocol such as UDP, a timestamp can be included in every datagram, so that the recipient can ignore datagrams that are too old or in the future. A timestamp is not secret data and doesn't have to be encrypted. Using the timestamp as additional data allows the recipient to confirm that a timestamp that appears to be valid hasn't been tampered with.

Ordered and unordered messages

As described above, a simple way to ensure that a sequence of received messages matches what the originator sent is to set the initial nonce to a given value, and increment it after every message.

The originator only sends the initialization vector. Individual messages do not contain a copy of the nonce used to encrypt them. They don't have to, since the recipient can perform the same operation as the sender, namely increment the nonce after every received encrypted message, in order to decrypt the stream.

If the stream being decrypted doesn't match the original stream, because messages have been altered, removed, added, duplicated or reordered, the authentication tag will not match the one computed by the recipient using the expected nonce. This issue will be immediately detected by the decryption function.

When using a transport protocol such as UDP, encrypted messages are not guaranteed to be received in order. Some datagrams may also be missing or duplicated. Applications must reorder them and handle retransmission.

In that situation, a copy of the nonce, or value representing the difference with the initial nonce, can be added to every encrypted message. Since a message is encrypted and authenticated using a unique nonce in addition to the key, the decryption process will immediately detect a an encrypted message whose attached nonce has been tampered with.

Shared keys and repeated nonces

As previously stated, it is important to avoid using the same nonce to encrypt different messages.

If two or more parties share the same secret key, increment the nonce after each message, but use the same initialization vector, different messages may end up being encrypted with the same nonce.

Each party can start with a different initialization vector and send it to its peers, but a better approach is to simply use different keys. Even if `kAB` and `kBA` are known by both parties, messages sent by `A` to `B` are encrypted using the secret key `kAB`, whereas messages sent by `B` to `A` will be encrypted using `kBA`.

The key exchange API (`crypto_kx()` functions) creates two different keys for that purpose.

Nonce-misuse resistance

Libsodium assumes a platform that can produce strong random numbers. On some embedded systems, this may not be the case, and in such a scenario, having a monotonically increasing, global counter is rarely a practical solution either.

In that scenario, nonces can be constructed as follows: `Hk(message) + message_counter`. This assumes nonces that are 160-bit long or more.

`Hk` is a keyed hash function safe against length-extension attacks, such as the one provided by `crypto_generichash()`, preferably used with a secret key that is distinct from the one used for encryption.

Addition can be performed using the `sodium_add()` function.

The security guarantees is weaker than when using a random initialization vector. Namely, two sequences of messages sharing the same prefix will produce the same encrypted stream prefix. Therefore, this scheme must only be used on platforms that cannot produce secure random numbers. A sound alternative is to use a library specifically designed for these platforms, such as [libhydrogen](#).

Short nonces

Ciphers such as `AES` do not feature nonces large enough to be randomly chosen without taking the risk of repeating a nonce.

More accurately, a single nonce shouldn't be used to encrypt different messages with the same key. Using the same nonce to encrypt different messages with different keys is perfectly safe.

Therefore, ciphers with short nonces can be safely used, but require keys to be frequently rotated, in addition to generating a new key for every stream. This requires support from application-level protocols, and can be tricky to implement.

An alternative is to use a nonce extension mechanism. A large (160 bits or more) nonce `N` is used by the protocol. Its initial value can be randomly chosen.

The actual encryption is done as follows:

- Using a pseudorandom function, a subkey and a shorter nonce are derived from the key and the large nonce
- The cipher is used with this subkey and short nonce to encrypt or decrypt a message

The following code snippet derives a 256-bit subkey and a 96-bit subnonce (these parameters can be used with AES-256) from a 256-bit key and an arbitrary long nonce:

```
unsigned char out[32 + 12];
unsigned char *subkey = out;
unsigned char *subnonce = out + 32;
crypto_generichash(out, sizeof out, nonce, sizeof nonce, key, sizeof key);
```

Note that the security of the cipher is reduced to the one of the hash function. This operation also implies a small performance hit, that becomes negligible as the message size increases.

Unless a cipher such as `AES` is a requirement, using a cipher with a longer nonce is easier and safer.

Note

Please refer to the [main page on AEAD constructions](#) for detailed information about the limitations of each construction.

Secret-key authentication

Example

```
#define MESSAGE (const unsigned char *) "test"
#define MESSAGE_LEN 4

unsigned char key[crypto_auth_KEYBYTES];
unsigned char mac[crypto_auth_BYTES];

crypto_auth_keygen(key);
crypto_auth(mac, MESSAGE, MESSAGE_LEN, key);

if (crypto_auth_verify(mac, MESSAGE, MESSAGE_LEN, key) != 0) {
    /* message forged! */
}
```

Purpose

This operation computes an authentication tag for a message and a secret key, and provides a way to verify that a given tag is valid for a given message and a key.

The function computing the tag deterministic: the same (message, key) tuple will always produce the same output.

However, even if the message is public, knowing the key is required in order to be able to compute a valid tag. Therefore, the key should remain confidential. The tag, however, can be public.

A typical use case is:

- A prepares a message, add an authentication tag, sends it to B
- A doesn't store the message
- Later on, B sends the message and the authentication tag to A
- A uses the authentication tag to verify that it created this message.

This operation does *not* encrypt the message. It only computes and verifies an authentication tag.

Usage


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

The `crypto_auth()` function computes a tag for the message `in`, whose length is `inlen` bytes, and the key `k`. `k` should be `crypto_auth_KEYBYTES` bytes. The function puts the tag into `out`. The tag is `crypto_auth_BYTES` bytes long.

```
int crypto_auth_verify(const unsigned char *h, const unsigned char *in,
                      unsigned long long inlen, const unsigned char *k);
```

The `crypto_auth_verify()` function verifies that the tag stored at `h` is a valid tag for the message `in` whose length is `inlen` bytes, and the key `k`.

It returns `-1` if the verification fails, and `0` if it passes.

```
void crypto_auth_keygen(unsigned char k[crypto_auth_KEYBYTES]);
```

This helper function introduced in libsodium 1.0.12 creates a random key `k`.

It is equivalent to calling `randombytes_buf()` but improves code clarity and can prevent misuse by ensuring that the provided key length is always be correct.

Constants

- `crypto_auth_BYTES`
- `crypto_auth_KEYBYTES`

Algorithm details

- HMAC-SHA512-256

Authenticated Encryption with Additional Data

This operation:

- Encrypts a message with a key and a nonce to keep it confidential
- Computes an authentication tag. This tag is used to make sure that the message, as well as optional, non-confidential (non-encrypted) data, haven't been tampered with.

A typical use case for additional data is to authenticate protocol-specific metadata about the message, such as its length and encoding.

Supported constructions

libsodium supports two popular constructions: AES256-GCM and ChaCha20-Poly1305 (original version and IETF version), as well as a variant of the later with an extended nonce: XChaCha20-Poly1305.

The "combined mode" API of each construction appends the authentication tag to the ciphertext. The "detached mode" API stores the authentication tag in a separate location.

Availability and interoperability

Construction	Key size	Nonce size	Block size	MAC size	Availability
AES256-GCM	256 bits	96 bits	128 bits	128 bits	libsodium \geq 1.0.4 but requires hardware support. IETF standard; also implemented in many other libraries.
ChaCha20-Poly1305	256 bits	64 bits	512 bits	128 bits	libsodium \geq 0.6.0. Also implemented in {Libre,Open,Boring}SSL.
ChaCha20-Poly1305-IETF	256 bits	96 bits	512 bits	128 bits	libsodium \geq 1.0.4. IETF standard; also implemented in Ring, {Libre,Open,Boring}SSL and other libraries.
XChaCha20-Poly1305-IETF	256 bits	192 bits	512 bits	128 bits	libsodium \geq 1.0.12. On the standard track.

Limitations

Construction	Max bytes for a single (key,nonce)	Max bytes for a single key
AES256-GCM	~ 64 GB	~ 350 GB (for ~16 KB long messages)
ChaCha20-Poly1305	No practical limits (~ 2^{64} bytes)	Up to 2^{64}^* messages, no practical total size limits
ChaCha20-Poly1305-IETF	256 GB	Up to 2^{64}^* messages, no practical total size limits
XChaCha20-Poly1305-IETF	No practical limits (~ 2^{64} bytes)	Up to 2^{64}^* messages, no practical total size limits

These figures assume an untruncated (128-bit) authentication tag.

* Although periodic rekeying remains highly recommended, online protocols leveraging additional data to discard old messages don't have practical limitations on the total number of messages.

Nonces

Construction	Safe options to choose a nonce
AES256-GCM	Counter, permutation
ChaCha20-Poly1305	Counter, permutation
ChaCha20-Poly1305-IETF	Counter, permutation
XChaCha20-Poly1305-IETF	Counter, permutation, random, $H_k(\text{random} \parallel m)$

H_k represents a keyed hash function that is safe against length-extension attacks, such as BLAKE2 or the HMAC construction.

TL;DR: which one should I use?

XChaCha20-Poly1305-IETF is the safest choice.

Other choices are only present for interoperability with other libraries that don't implement

XChaCha20-Poly1305-IETF yet.

AES256-GCM

The current implementation of this construction is hardware-accelerated and requires the Intel SSSE3 extensions, as well as the `aesni` and `pclmul` instructions.

Intel Westmere processors (introduced in 2010) and newer meet the requirements.

There are no plans to support non hardware-accelerated implementations of AES-GCM, as correctly mitigating side-channels in a software implementation comes with major speed tradeoffs, that defeat the whole point of AES-GCM over ChaCha20-Poly1305.

ChaCha20-Poly1305

While AES is very fast on dedicated hardware, its performance on platforms that lack such hardware is considerably lower. Another problem is that many software AES implementations are vulnerable to cache-collision timing attacks.

ChaCha20 is considerably faster than AES in software-only implementations, making it around three times as fast on platforms that lack specialized AES hardware. ChaCha20 is also not sensitive to timing attacks.

Poly1305 is a high-speed message authentication code.

The combination of the ChaCha20 stream cipher with the Poly1305 authenticator was proposed in January 2014 as an alternative to the Salsa20-Poly1305 construction. ChaCha20-Poly1305 was implemented in major operating systems, web browsers and crypto libraries shortly after. It eventually became an official IETF standard in May 2015.

The ChaCha20-Poly1305 implementation in libsodium is portable across all supported architectures.

XChaCha20-Poly1305

XChaCha20-Poly1305 applies the construction described in Daniel Bernstein's [Extending the Salsa20 nonce](#) paper to the ChaCha20 cipher in order to extend the nonce size to 192-bit.

This extended nonce size allows random nonces to be safely used, and also facilitates the construction of misuse-resistant schemes.

The XChaCha20-Poly1305 implementation in libsodium is portable across all supported architectures.

The main limitation of XChaCha20-Poly1305 is that it is not widely implemented in other libraries yet. However, it will [soon](#) become an IETF standard.

Additional data

These functions accept an optional, arbitrary long "additional data" parameter. These data are not present in the ciphertext, but are mixed in the computation of the authentication tag.

A typical use for these data is to authenticate version numbers, timestamps or monotonically increasing counters in order to discard previous messages and prevent replay attacks.

References

- [Limits on Authenticated Encryption Use in TLS](#) (Atul Luykx, Kenneth G. Paterson).

Authenticated Encryption with Additional Data using ChaCha20-Poly1305

Purpose

This operation:

- Encrypts a message with a key and a nonce to keep it confidential
- Computes an authentication tag. This tag is used to make sure that the message, as well as optional, non-confidential (non-encrypted) data, haven't been tampered with.

A typical use case for additional data is to store protocol-specific metadata about the message, such as its length and encoding.

The chosen construction uses encrypt-then-MAC and decryption will never be performed, even partially, before verification.

Variants

libsodium implements three versions of the ChaCha20-Poly1305 construction:

- The original construction can safely encrypt up to 2^{64} messages with the same key (even more with most protocols), without any practical limit to the size of a message (up to 2^{64} bytes for a 128-bit tag).
- The IETF variant. It can safely encrypt a practically unlimited number of messages, but individual messages cannot exceed $64 \cdot (2^{32}) - 64$ bytes (approximately 256 GB).
- The XChaCha20 variant, introduced in libsodium 1.0.12. It can safely encrypt a practically unlimited number of messages of any sizes, and random nonces are safe to use.

The first two variants are fully interoperable with other crypto libraries. The XChaCha20 variant is currently only implemented in libsodium, but is the recommended option if interoperability is not a concern.

They all share the same security properties when used properly, and are accessible via a similar API.

The `crypto_aead_chacha20poly1305_*` set of functions implements the original construction, the `crypto_aead_chacha20poly1305_ietf_*` functions implement the IETF version, and the `crypto_aead_xchacha20poly1305_ietf_*` functions implement the XChaCha20 variant.

The constants are the same, except for the nonce size.

The original ChaCha20-Poly1305 construction

The original ChaCha20-Poly1305 construction can safely encrypt a practically unlimited number of messages with the same key, without any practical limit to the size of a message (up to $\sim 2^{64}$ bytes).

Example (combined mode)

```
#define MESSAGE (const unsigned char *) "test"
#define MESSAGE_LEN 4
#define ADDITIONAL_DATA (const unsigned char *) "123456"
#define ADDITIONAL_DATA_LEN 6

unsigned char nonce[crypto_aead_chacha20poly1305_NPUBBYTES];
unsigned char key[crypto_aead_chacha20poly1305_KEYBYTES];
unsigned char ciphertext[MESSAGE_LEN + crypto_aead_chacha20poly1305_ABYTES];
unsigned long long ciphertext_len;

crypto_aead_chacha20poly1305_keygen(key);
randombytes_buf(nonce, sizeof nonce);

crypto_aead_chacha20poly1305_encrypt(ciphertext, &ciphertext_len,
                                     MESSAGE, MESSAGE_LEN,
                                     ADDITIONAL_DATA, ADDITIONAL_DATA_LEN,
                                     NULL, nonce, key);

unsigned char decrypted[MESSAGE_LEN];
unsigned long long decrypted_len;
if (crypto_aead_chacha20poly1305_decrypt(decrypted, &decrypted_len,
                                         NULL,
                                         ciphertext, ciphertext_len,
                                         ADDITIONAL_DATA,
                                         ADDITIONAL_DATA_LEN,
                                         nonce, key) != 0) {

    /* message forged! */
}
```

Combined mode

In combined mode, the authentication tag is directly appended to the encrypted message. This is usually what you want.


```
int crypto_aead_chacha20poly1305_encrypt(unsigned char *c,
                                         unsigned long long *clen_p,
                                         const unsigned char *m,
                                         unsigned long long mlen,
                                         const unsigned char *ad,
                                         unsigned long long adlen,
                                         const unsigned char *nsec,
                                         const unsigned char *npub,
                                         const unsigned char *k);
```

The `crypto_aead_chacha20poly1305_encrypt()` function encrypts a message `m` whose length is `mlen` bytes using a secret key `k` (`crypto_aead_chacha20poly1305_KEYBYTES` bytes) and public nonce `npub` (`crypto_aead_chacha20poly1305_NPUBBYTES` bytes).

The encrypted message, as well as a tag authenticating both the confidential message `m` and `adlen` bytes of non-confidential data `ad`, are put into `c`.

`ad` can be a `NULL` pointer with `adlen` equal to `0` if no additional data are required.

At most `mlen + crypto_aead_chacha20poly1305_ABYTES` bytes are put into `c`, and the actual number of bytes is stored into `clen` unless `clen` is a `NULL` pointer.

`nsec` is not used by this particular construction and should always be `NULL`.

The public nonce `npub` should never ever be reused with the same key. The recommended way to generate it is to use `randombytes_buf()` for the first message, and increment it for each subsequent message using the same key.

```
int crypto_aead_chacha20poly1305_decrypt(unsigned char *m,
                                         unsigned long long *mlen_p,
                                         unsigned char *nsec,
                                         const unsigned char *c,
                                         unsigned long long clen,
                                         const unsigned char *ad,
                                         unsigned long long adlen,
                                         const unsigned char *npub,
                                         const unsigned char *k);
```

The `crypto_aead_chacha20poly1305_decrypt()` function verifies that the ciphertext `c` (as produced by `crypto_aead_chacha20poly1305_encrypt()`) includes a valid tag using a secret key `k`, a public nonce `npub`, and additional data `ad` (`adlen` bytes).

`ad` can be a `NULL` pointer with `adlen` equal to `0` if no additional data are required.

`nsec` is not used by this particular construction and should always be `NULL`.

The function returns `-1` if the verification fails.

If the verification succeeds, the function returns `0`, puts the decrypted message into `m` and stores its actual number of bytes into `mlen` if `mlen` is not a `NULL` pointer.

At most `clen - crypto_aead_chacha20poly1305_ABYTES` bytes will be put into `m`.

Detached mode

Some applications may need to store the authentication tag and the encrypted message at different locations.

For this specific use case, "detached" variants of the functions above are available.

```
int crypto_aead_chacha20poly1305_encrypt_detached(unsigned char *c,
                                                    unsigned char *mac,
                                                    unsigned long long *maclen_p,
                                                    const unsigned char *m,
                                                    unsigned long long mlen,
                                                    const unsigned char *ad,
                                                    unsigned long long adlen,
                                                    const unsigned char *nsec,
                                                    const unsigned char *npub,
                                                    const unsigned char *k);
```

The `crypto_aead_chacha20poly1305_encrypt_detached()` function encrypts a message `m` with a key `k` and a nonce `npub`. It puts the resulting ciphertext, whose length is equal to the message, into `c`.

It also computes a tag that authenticates the ciphertext as well as optional, additional data `ad` of length `adlen`. This tag is put into `mac`, and its length is `crypto_aead_chacha20poly1305_ABYTES` bytes.

`nsec` is not used by this particular construction and should always be `NULL`.

```
int crypto_aead_chacha20poly1305_decrypt_detached(unsigned char *m,
                                                    unsigned char *nsec,
                                                    const unsigned char *c,
                                                    unsigned long long clen,
                                                    const unsigned char *mac,
                                                    const unsigned char *ad,
                                                    unsigned long long adlen,
                                                    const unsigned char *npub,
                                                    const unsigned char *k);
```

The `crypto_aead_chacha20poly1305_decrypt_detached()` function verifies that the authentication tag `mac` is valid for the ciphertext `c` of length `clen` bytes, the key `k`, the nonce `npub` and optional, additional data `ad` of length `adlen` bytes.

If the tag is not valid, the function returns `-1` and doesn't do any further processing.

If the tag is valid, the ciphertext is decrypted and the plaintext is put into `m`. The length is equal to the length of the ciphertext.

`nsec` is not used by this particular construction and should always be `NULL`.

```
void crypto_aead_chacha20poly1305_keygen(unsigned char k[crypto_aead_chacha20poly1305_
KEYBYTES]);
```

This helper function introduced in libsodium 1.0.12 creates a random key `k`.

It is equivalent to calling `randombytes_buf()` but improves code clarity and can prevent misuse by ensuring that the provided key length is always be correct.

Constants

- `crypto_aead_chacha20poly1305_KEYBYTES`
- `crypto_aead_chacha20poly1305_NPUBBYTES`
- `crypto_aead_chacha20poly1305_ABYTES`

Algorithm details

- Encryption: ChaCha20 stream cipher
- Authentication: Poly1305 MAC

Notes

In order to prevent nonce reuse, if a key is being reused, it is recommended to increment the previous nonce instead of generating a random nonce for each message.

To prevent nonce reuse in a client-server protocol, either use different keys for each direction, or make sure that a bit is masked in one direction, and set in the other.

See also

- [ChaCha20 and Poly1305 based Cipher Suites for TLS](#) - Specification of the original construction

The IETF ChaCha20-Poly1305 construction

The IETF variant of the ChaCha20-Poly1305 construction can safely encrypt a practically unlimited number of messages, but individual messages cannot exceed $64 \cdot (2^{32}) - 64$ bytes (approximatively 256 GiB).

Example (combined mode)

```
#define MESSAGE (const unsigned char *) "test"
#define MESSAGE_LEN 4
#define ADDITIONAL_DATA (const unsigned char *) "123456"
#define ADDITIONAL_DATA_LEN 6

unsigned char nonce[crypto_aead_chacha20poly1305_IETF_NPUBBYTES];
unsigned char key[crypto_aead_chacha20poly1305_IETF_KEYBYTES];
unsigned char ciphertext[MESSAGE_LEN + crypto_aead_chacha20poly1305_IETF_ABYTES];
unsigned long long ciphertext_len;

crypto_aead_chacha20poly1305_ietf_keygen(key);
randombytes_buf(nonce, sizeof nonce);

crypto_aead_chacha20poly1305_ietf_encrypt(ciphertext, &ciphertext_len,
                                         MESSAGE, MESSAGE_LEN,
                                         ADDITIONAL_DATA, ADDITIONAL_DATA_LEN,
                                         NULL, nonce, key);

unsigned char decrypted[MESSAGE_LEN];
unsigned long long decrypted_len;
if (crypto_aead_chacha20poly1305_ietf_decrypt(decrypted, &decrypted_len,
                                             NULL,
                                             ciphertext, ciphertext_len,
                                             ADDITIONAL_DATA,
                                             ADDITIONAL_DATA_LEN,
                                             nonce, key) != 0) {

    /* message forged! */
}
```

Combined mode

In combined mode, the authentication tag is directly appended to the encrypted message. This is usually what you want.

```
int crypto_aead_chacha20poly1305_ietf_encrypt(unsigned char *c,
                                              unsigned long long *clen_p,
                                              const unsigned char *m,
                                              unsigned long long mlen,
                                              const unsigned char *ad,
                                              unsigned long long adlen,
                                              const unsigned char *nsec,
                                              const unsigned char *npub,
                                              const unsigned char *k);
```

The `crypto_aead_chacha20poly1305_ietf_encrypt()` function encrypts a message `m` whose length is `mlen` bytes using a secret key `k` (`crypto_aead_chacha20poly1305_IETF_KEYBYTES` bytes) and public nonce `npub` (`crypto_aead_chacha20poly1305_IETF_NPUBBYTES` bytes).

The encrypted message, as well as a tag authenticating both the confidential message `m` and `adlen` bytes of non-confidential data `ad` , are put into `c` .

`ad` can be a `NULL` pointer with `adlen` equal to `0` if no additional data are required.

At most `mlen + crypto_aead_chacha20poly1305_IETF_ABYTES` bytes are put into `c` , and the actual number of bytes is stored into `clen` unless `clen` is a `NULL` pointer.

`nsec` is not used by this particular construction and should always be `NULL` .

The public nonce `npub` should never ever be reused with the same key. The recommended way to generate it is to use `randombytes_buf()` for the first message, and increment it for each subsequent message using the same key.

```
int crypto_aead_chacha20poly1305_ietf_decrypt(unsigned char *m,
                                              unsigned long long *mlen_p,
                                              unsigned char *nsec,
                                              const unsigned char *c,
                                              unsigned long long clen,
                                              const unsigned char *ad,
                                              unsigned long long adlen,
                                              const unsigned char *npub,
                                              const unsigned char *k);
```

The `crypto_aead_chacha20poly1305_ietf_decrypt()` function verifies that the ciphertext `c` (as produced by `crypto_aead_chacha20poly1305_ietf_encrypt()`) includes a valid tag using a secret key `k` , a public nonce `npub` , and additional data `ad` (`adlen` bytes).

`ad` can be a `NULL` pointer with `adlen` equal to `0` if no additional data are required.

`nsec` is not used by this particular construction and should always be `NULL` .

The function returns `-1` if the verification fails.

If the verification succeeds, the function returns `0`, puts the decrypted message into `m` and stores its actual number of bytes into `mlen` if `mlen` is not a `NULL` pointer.

At most `clen - crypto_aead_chacha20poly1305_IETF_ABYTES` bytes will be put into `m`.

Detached mode

Some applications may need to store the authentication tag and the encrypted message at different locations.

For this specific use case, "detached" variants of the functions above are available.

```
int crypto_aead_chacha20poly1305_ietf_encrypt_detached(unsigned char *c,
                                                         unsigned char *mac,
                                                         unsigned long long *maclen_p,
                                                         const unsigned char *m,
                                                         unsigned long long mlen,
                                                         const unsigned char *ad,
                                                         unsigned long long adlen,
                                                         const unsigned char *nsec,
                                                         const unsigned char *npub,
                                                         const unsigned char *k);
```

The `crypto_aead_chacha20poly1305_ietf_encrypt_detached()` function encrypts a message `m` with a key `k` and a nonce `npub`. It puts the resulting ciphertext, whose length is equal to the message, into `c`.

It also computes a tag that authenticates the ciphertext as well as optional, additional data `ad` of length `adlen`. This tag is put into `mac`, and its length is `crypto_aead_chacha20poly1305_IETF_ABYTES` bytes.

`nsec` is not used by this particular construction and should always be `NULL`.

```
int crypto_aead_chacha20poly1305_ietf_decrypt_detached(unsigned char *m,
                                                         unsigned char *nsec,
                                                         const unsigned char *c,
                                                         unsigned long long clen,
                                                         const unsigned char *mac,
                                                         const unsigned char *ad,
                                                         unsigned long long adlen,
                                                         const unsigned char *npub,
                                                         const unsigned char *k);
```

The `crypto_aead_chacha20poly1305_ietf_decrypt_detached()` function verifies that the authentication tag `mac` is valid for the ciphertext `c` of length `clen` bytes, the key `k`, the nonce `npub` and optional, additional data `ad` of length `adlen` bytes.

If the tag is not valid, the function returns `-1` and doesn't do any further processing.

If the tag is valid, the ciphertext is decrypted and the plaintext is put into `m`. The length is equal to the length of the ciphertext.

`nsec` is not used by this particular construction and should always be `NULL`.

```
void crypto_aead_chacha20poly1305_ietf_keygen(unsigned char k[crypto_aead_chacha20poly1305_ietf_KEYBYTES]);
```

This helper function introduced in libsodium 1.0.12 creates a random key `k`.

It is equivalent to calling `randombytes_buf()` but improves code clarity and can prevent misuse by ensuring that the provided key length is always be correct.

Constants

- `crypto_aead_chacha20poly1305_IETF_ABYTES`
- `crypto_aead_chacha20poly1305_IETF_KEYBYTES`
- `crypto_aead_chacha20poly1305_IETF_NPUBBYTES`

On earlier versions, use `crypto_aead_chacha20poly1305_KEYBYTES` and `crypto_aead_chacha20poly1305_NPUBBYTES` - The nonce size is the only constant that differs between the original variant and the IETF variant.

Algorithm details

- Encryption: ChaCha20 stream cipher
- Authentication: Poly1305 MAC

Notes

In order to prevent nonce reuse, if a key is being reused, it is recommended to increment the previous nonce instead of generating a random nonce for each message.

To prevent nonce reuse in a client-server protocol, either use different keys for each direction, or make sure that a bit is masked in one direction, and set in the other.

See also

- [ChaCha20 and Poly1305 for IETF protocols](#) - Specification of the IETF variant

The XChaCha20-Poly1305 construction

The XChaCha20-Poly1305 construction can safely encrypt a practically unlimited number of messages with the same key, without any practical limit to the size of a message (up to $\sim 2^{64}$ bytes).

As an alternative to counters, its large nonce size (192-bit) allows random nonces to be safely used.

For this reason, and if interoperability with other libraries is not a concern, this is the recommended AEAD construction.

Example (combined mode)

```
#define MESSAGE (const unsigned char *) "test"
#define MESSAGE_LEN 4
#define ADDITIONAL_DATA (const unsigned char *) "123456"
#define ADDITIONAL_DATA_LEN 6

unsigned char nonce[crypto_aead_xchacha20poly1305_ietf_NPUBBYTES];
unsigned char key[crypto_aead_xchacha20poly1305_ietf_KEYBYTES];
unsigned char ciphertext[MESSAGE_LEN + crypto_aead_xchacha20poly1305_ietf_ABYTES];
unsigned long long ciphertext_len;

crypto_aead_xchacha20poly1305_ietf_keygen(key);
randombytes_buf(nonce, sizeof nonce);

crypto_aead_xchacha20poly1305_ietf_encrypt(ciphertext, &ciphertext_len,
                                           MESSAGE, MESSAGE_LEN,
                                           ADDITIONAL_DATA, ADDITIONAL_DATA_LEN,
                                           NULL, nonce, key);

unsigned char decrypted[MESSAGE_LEN];
unsigned long long decrypted_len;
if (crypto_aead_xchacha20poly1305_ietf_decrypt(decrypted, &decrypted_len,
                                              NULL,
                                              ciphertext, ciphertext_len,
                                              ADDITIONAL_DATA,
                                              ADDITIONAL_DATA_LEN,
                                              nonce, key) != 0) {

    /* message forged! */
}
```

Combined mode

In combined mode, the authentication tag is directly appended to the encrypted message. This is usually what you want.

```
int crypto_aead_xchacha20poly1305_ietf_encrypt(unsigned char *c,
                                                unsigned long long *clen_p,
                                                const unsigned char *m,
                                                unsigned long long mlen,
                                                const unsigned char *ad,
                                                unsigned long long adlen,
                                                const unsigned char *nsec,
                                                const unsigned char *npub,
                                                const unsigned char *k);
```

The `crypto_aead_xchacha20poly1305_ietf_encrypt()` function encrypts a message `m` whose length is `mlen` bytes using a secret key `k` (`crypto_aead_xchacha20poly1305_ietf_KEYBYTES` bytes) and public nonce `npub` (`crypto_aead_xchacha20poly1305_ietf_NPUBBYTES` bytes).

The encrypted message, as well as a tag authenticating both the confidential message `m` and `adlen` bytes of non-confidential data `ad`, are put into `c`.

`ad` can be a `NULL` pointer with `adlen` equal to `0` if no additional data are required.

At most `mlen + crypto_aead_xchacha20poly1305_ietf_ABYTES` bytes are put into `c`, and the actual number of bytes is stored into `clen` unless `clen` is a `NULL` pointer.

`nsec` is not used by this particular construction and should always be `NULL`.

The public nonce `npub` should never ever be reused with the same key. Nonces can be generated using `randombytes_buf()` for every message. XChaCha20 uses 192-bit nonces, so the probability of a collision is negligible. Using a counter is also perfectly fine: nonces have to be unique for a given key, but they don't have to be unpredictable.

```
int crypto_aead_xchacha20poly1305_ietf_decrypt(unsigned char *m,
                                                unsigned long long *mlen_p,
                                                unsigned char *nsec,
                                                const unsigned char *c,
                                                unsigned long long clen,
                                                const unsigned char *ad,
                                                unsigned long long adlen,
                                                const unsigned char *npub,
                                                const unsigned char *k);
```

The `crypto_aead_xchacha20poly1305_ietf_decrypt()` function verifies that the ciphertext `c` (as produced by `crypto_aead_xchacha20poly1305_ietf_encrypt()`) includes a valid tag using a secret key `k`, a public nonce `npub`, and additional data `ad` (`adlen` bytes).

`ad` can be a `NULL` pointer with `adlen` equal to `0` if no additional data are required.

`nsec` is not used by this particular construction and should always be `NULL`.

The function returns `-1` if the verification fails.

If the verification succeeds, the function returns `0`, puts the decrypted message into `m` and stores its actual number of bytes into `mlen` if `m` is not a `NULL` pointer.

At most `clen - crypto_aead_xchacha20poly1305_ietf_ABYTES` bytes will be put into `m`.

Detached mode

Some applications may need to store the authentication tag and the encrypted message at different locations.

For this specific use case, "detached" variants of the functions above are available.

```
int crypto_aead_xchacha20poly1305_ietf_encrypt_detached(unsigned char *c,
                                                         unsigned char *mac,
                                                         unsigned long long *maclen_p,
                                                         const unsigned char *m,
                                                         unsigned long long mlen,
                                                         const unsigned char *ad,
                                                         unsigned long long adlen,
                                                         const unsigned char *nsec,
                                                         const unsigned char *npub,
                                                         const unsigned char *k);
```

The `crypto_aead_xchacha20poly1305_ietf_encrypt_detached()` function encrypts a message `m` with a key `k` and a nonce `npub`. It puts the resulting ciphertext, whose length is equal to the message, into `c`.

It also computes a tag that authenticates the ciphertext as well as optional, additional data `ad` of length `adlen`. This tag is put into `mac`, and its length is `crypto_aead_xchacha20poly1305_ietf_ABYTES` bytes.

`nsec` is not used by this particular construction and should always be `NULL`.

```
int crypto_aead_xchacha20poly1305_ietf_decrypt_detached(unsigned char *m,
                                                         unsigned char *nsec,
                                                         const unsigned char *c,
                                                         unsigned long long clen,
                                                         const unsigned char *mac,
                                                         const unsigned char *ad,
                                                         unsigned long long adlen,
                                                         const unsigned char *npub,
                                                         const unsigned char *k);
```

The `crypto_aead_xchacha20poly1305_ietf_decrypt_detached()` function verifies that the authentication tag `mac` is valid for the ciphertext `c` of length `clen` bytes, the key `k`, the nonce `npub` and optional, additional data `ad` of length `adlen` bytes.

If the tag is not valid, the function returns `-1` and doesn't do any further processing.

If the tag is valid, the ciphertext is decrypted and the plaintext is put into `m`. The length is equal to the length of the ciphertext.

`nsec` is not used by this particular construction and should always be `NULL`.

```
void crypto_aead_xchacha20poly1305_ietf_keygen(unsigned char k[crypto_aead_xchacha20poly1305_ietf_KEYBYTES]);
```

This helper function introduced in libsodium 1.0.12 creates a random key `k`.

It is equivalent to calling `randombytes_buf()` but improves code clarity and can prevent misuse by ensuring that the provided key length is always be correct.

Constants

- `crypto_aead_xchacha20poly1305_ietf_KEYBYTES`
- `crypto_aead_xchacha20poly1305_ietf_NPUBBYTES`
- `crypto_aead_xchacha20poly1305_ietf_ABYTES`

Algorithm details

- Encryption: XChaCha20 stream cipher
- Authentication: Poly1305 MAC

Notes

XChaCha20-Poly1305 was introduced in libsodium 1.0.12.

Unlike other variants directly using the ChaCha20 cipher, generating a random nonce for each message is acceptable with this XChaCha20-based construction, provided that the output of the PRNG is indistinguishable from random data.

Authenticated Encryption with Additional Data using AES-GCM

Warning

WARNING: Despite being the most popular AEAD construction due to its use in TLS, safely using AES-GCM in a different context is tricky.

No more than ~ 350 GB of input data should be encrypted with a given key. This is for ~ 16 KB messages -- Actual figures vary according to message sizes.

In addition, nonces are short and repeated nonces would totally destroy the security of this scheme. Nonces should thus come from atomic counters, which can be difficult to set up in a distributed environment.

Unless you absolutely need AES-GCM, use XChaCha20-Poly1305

(`crypto_aead_xchacha20poly1305_ietf_*`()) instead. It doesn't have any of these limitations.

Or, if you don't need to authenticate additional data, just stick to `crypto_secretbox()` .

Example (combined mode)

```

#include <sodium.h>

#define MESSAGE (const unsigned char *) "test"
#define MESSAGE_LEN 4
#define ADDITIONAL_DATA (const unsigned char *) "123456"
#define ADDITIONAL_DATA_LEN 6

unsigned char nonce[crypto_aead_aes256gcm_NPUBBYTES];
unsigned char key[crypto_aead_aes256gcm_KEYBYTES];
unsigned char ciphertext[MESSAGE_LEN + crypto_aead_aes256gcm_ABYTES];
unsigned long long ciphertext_len;

sodium_init();
if (crypto_aead_aes256gcm_is_available() == 0) {
    abort(); /* Not available on this CPU */
}

crypto_aead_aes256gcm_keygen(key);
randombytes_buf(nonce, sizeof nonce);

crypto_aead_aes256gcm_encrypt(ciphertext, &ciphertext_len,
                              MESSAGE, MESSAGE_LEN,
                              ADDITIONAL_DATA, ADDITIONAL_DATA_LEN,
                              NULL, nonce, key);

unsigned char decrypted[MESSAGE_LEN];
unsigned long long decrypted_len;
if (ciphertext_len < crypto_aead_aes256gcm_ABYTES ||
    crypto_aead_aes256gcm_decrypt(decrypted, &decrypted_len,
                                   NULL,
                                   ciphertext, ciphertext_len,
                                   ADDITIONAL_DATA,
                                   ADDITIONAL_DATA_LEN,
                                   nonce, key) != 0) {
    /* message forged! */
}

```

Purpose

This operation:

- Encrypts a message with a key and a nonce to keep it confidential
- Computes an authentication tag. This tag is used to make sure that the message, as well as optional, non-confidential (non-encrypted) data, haven't been tampered with.

A typical use case for additional data is to store protocol-specific metadata about the message, such as its length and encoding.

It can also be used as a MAC, with an empty message.

Decryption will never be performed, even partially, before verification.

When supported by the CPU, AES-GCM is the fastest AEAD cipher available in this library.

Limitations

The current implementation of this construction is hardware-accelerated and requires the Intel SSSE3 extensions, as well as the `aesni` and `pclmul` instructions.

Intel Westmere processors (introduced in 2010) and newer meet the requirements.

There are no plans to support non hardware-accelerated implementations of AES-GCM. If portability is a concern, use ChaCha20-Poly1305 instead.

Before using the functions below, hardware support for AES can be checked with:

```
int crypto_aead_aes256gcm_is_available(void);
```

The function returns `1` if the current CPU supports the AES256-GCM implementation, and `0` if it doesn't.

The library must have been initialized with `sodium_init()` prior to calling this function.

Combined mode

In combined mode, the authentication tag is directly appended to the encrypted message. This is usually what you want.

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

The function `crypto_aead_aes256gcm_encrypt()` encrypts a message `m` whose length is `mlen` bytes using a secret key `k` (`crypto_aead_aes256gcm_KEYBYTES` bytes) and a public nonce `npub` (`crypto_aead_aes256gcm_NPUBBYTES` bytes).

The encrypted message, as well as a tag authenticating both the confidential message `m` and `adlen` bytes of non-confidential data `ad`, are put into `c`.

`ad` can also be a `NULL` pointer if no additional data are required.

At most `milen + crypto_aead_aes256gcm_ABYTES` bytes are put into `c`, and the actual number of bytes is stored into `clen` if `clen` is not a `NULL` pointer.

`nsec` is not used by this particular construction and should always be `NULL`.

The function always returns `0`.

The public nonce `npub` should never ever be reused with the same key. The recommended way to generate it is to use `randombytes_buf()` for the first message, and then to increment it for each subsequent message using the same key.

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

The function `crypto_aead_aes256gcm_decrypt()` verifies that the ciphertext `c` (as produced by `crypto_aead_aes256gcm_encrypt()`), includes a valid tag using a secret key `k`, a public nonce `npub`, and additional data `ad` (`adlen` bytes). `clen` is the ciphertext length in bytes with the authenticator, so it has to be at least `ae256gcm_ABYTES`.

`ad` can be a `NULL` pointer if no additional data are required.

`nsec` is not used by this particular construction and should always be `NULL`.

The function returns `-1` if the verification fails.

If the verification succeeds, the function returns `0`, puts the decrypted message into `m` and stores its actual number of bytes into `milen` if `milen` is not a `NULL` pointer.

At most `clen - crypto_aead_aes256gcm_ABYTES` bytes will be put into `m`.

Detached mode

Some applications may need to store the authentication tag and the encrypted message at different locations.

For this specific use case, "detached" variants of the functions above are available.

```
int crypto_aead_aes256gcm_encrypt_detached(unsigned char *c,
                                           unsigned char *mac,
                                           unsigned long long *maclen_p,
                                           const unsigned char *m,
                                           unsigned long long mlen,
                                           const unsigned char *ad,
                                           unsigned long long adlen,
                                           const unsigned char *nsec,
                                           const unsigned char *npub,
                                           const unsigned char *k);
```

`crypto_aead_aes256gcm_encrypt_detached()` encrypts a message `m` whose length is `mlen` bytes using a secret key `k` (`crypto_aead_aes256gcm_KEYBYTES` bytes) and a public nonce `npub` (`crypto_aead_aes256gcm_NPUBBYTES` bytes).

The encrypted message is put into `c` . A tag authenticating both the confidential message `m` and `adlen` bytes of non-confidential data `ad` is put into `mac` .

`ad` can also be a `NULL` pointer if no additional data are required.

`crypto_aead_aes256gcm_ABYTES` bytes are put into `mac` , and the actual number of bytes required for verification is stored into `maclen_p` , unless `maclen_p` is `NULL` pointer.

`nsec` is not used by this particular construction and should always be `NULL` .

The function always returns `0` .

```
int crypto_aead_aes256gcm_decrypt_detached(unsigned char *m,
                                           unsigned char *nsec,
                                           const unsigned char *c,
                                           unsigned long long clen,
                                           const unsigned char *mac,
                                           const unsigned char *ad,
                                           unsigned long long adlen,
                                           const unsigned char *npub,
                                           const unsigned char *k);
```

The function `crypto_aead_aes256gcm_decrypt_detached()` verifies that the tag `mac` is valid for the ciphertext `c` using a secret key `k` , a public nonce `npub` , and additional data `ad` (`adlen` bytes).

`clen` is the ciphertext length in bytes.

`ad` can be a `NULL` pointer if no additional data are required.

`nsec` is not used by this particular construction and should always be `NULL` .

The function returns `-1` if the verification fails.

If the verification succeeds, the function returns `0`, and puts the decrypted message into `m`, whose length is equal to the length of the ciphertext.

```
void crypto_aead_aes256gcm_keygen(unsigned char k[crypto_aead_aes256gcm_KEYBYTES]);
```

This helper function introduced in libsodium 1.0.12 creates a random key `k`.

It is equivalent to calling `randombytes_buf()` but improves code clarity and can prevent misuse by ensuring that the provided key length is always be correct.

Constants

- `crypto_aead_aes256gcm_KEYBYTES`
- `crypto_aead_aes256gcm_NPUBBYTES`
- `crypto_aead_aes256gcm_ABYTES`

Notes

The nonce is 96 bits long. In order to prevent nonce reuse, if a key is being reused, it is recommended to increment the previous nonce instead of generating a random nonce for each message.

To prevent nonce reuse in a client-server protocol, either use different keys for each direction, or make sure that a bit is masked in one direction, and set in the other. The `crypto_kx_*` API can be used to do so.

When using AES-GCM, it is also recommended to switch to a new key before reaching ~350 GB encrypted with the same key. If frequent rekeying is not an option, use (X)ChaCha20-Poly1305 instead.

AES256-GCM with precomputation

Applications that encrypt several messages using the same key can gain a little speed by expanding the AES key only once, via the precalculation interface.

```
int crypto_aead_aes256gcm_beforenm(crypto_aead_aes256gcm_state *ctx_,
                                   const unsigned char *k);
```

The `crypto_aead_aes256gcm_beforenm()` function initializes a context `ctx` by expanding the key `k` and always returns `0`.

A 16 bytes alignment is required for the address of `ctx`. The size of this value can be obtained using `sizeof(crypto_aead_aes256gcm_state)`, or `crypto_aead_aes256gcm_statebytes()`.

Combined mode with precalculation

```
int crypto_aead_aes256gcm_encrypt_afternm(unsigned char *c,
                                           unsigned long long *clen_p,
                                           const unsigned char *m,
                                           unsigned long long mlen,
                                           const unsigned char *ad,
                                           unsigned long long adlen,
                                           const unsigned char *nsec,
                                           const unsigned char *npub,
                                           const crypto_aead_aes256gcm_state *ctx_);
```

```
int crypto_aead_aes256gcm_decrypt_afternm(unsigned char *m,
                                           unsigned long long *mlen_p,
                                           unsigned char *nsec,
                                           const unsigned char *c,
                                           unsigned long long clen,
                                           const unsigned char *ad,
                                           unsigned long long adlen,
                                           const unsigned char *npub,
                                           const crypto_aead_aes256gcm_state *ctx_);
```

The `crypto_aead_aes256gcm_encrypt_afternm()` and `crypto_aead_aes256gcm_decrypt_afternm()` functions are identical to `crypto_aead_aes256gcm_encrypt()` and `crypto_aead_aes256gcm_decrypt()`, but accept a previously initialized context `ctx` instead of a key.

Detached mode with precalculation

```
int crypto_aead_aes256gcm_encrypt_detached_afternm(unsigned char *c,
                                                    unsigned char *mac,
                                                    unsigned long long *maclen_p,
                                                    const unsigned char *m,
                                                    unsigned long long mlen,
                                                    const unsigned char *ad,
                                                    unsigned long long adlen,
                                                    const unsigned char *nsec,
                                                    const unsigned char *npub,
                                                    const crypto_aead_aes256gcm_state *
ctx_);
```

```
int crypto_aead_aes256gcm_decrypt_detached_afternm(unsigned char *m,
                                                    unsigned char *nsec,
                                                    const unsigned char *c,
                                                    unsigned long long clen,
                                                    const unsigned char *mac,
                                                    const unsigned char *ad,
                                                    unsigned long long adlen,
                                                    const unsigned char *npub,
                                                    const crypto_aead_aes256gcm_state *
ctx_)
```

The `crypto_aead_aes256gcm_encrypt_detached_afternm()` and `crypto_aead_aes256gcm_decrypt_detached_afternm()` functions are identical to `crypto_aead_aes256gcm_encrypt_detached()` and `crypto_aead_aes256gcm_decrypt_detached()` , but accept a previously initialized context `ctx` instead of a key.

Constants

- `crypto_aead_aes256gcm_KEYBYTES`
- `crypto_aead_aes256gcm_NPUBBYTES`
- `crypto_aead_aes256gcm_ABYTES`

Data types

- `crypto_aead_aes256gcm_state`

Notes

The nonce is 96 bits long. In order to prevent nonce reuse, if a key is being reused, it is recommended to increment the previous nonce instead of generating a random nonce for each message. To prevent nonce reuse in a client-server protocol, either use different keys for each direction, or make sure that a bit is masked in one direction, and set in the other.

When using AES-GCM, it is also recommended to switch to a new key before reaching ~350 GB encrypted with the same key. If frequent rekeying is not an option, use (X)ChaCha20-Poly1305 instead.

Public-key cryptography

Public-key cryptography refers to cryptographic systems that require two different keys, linked together by some one-way mathematical relationship (which depends on the algorithm used, but in any case the private key may never be recovered from the public key).

Public-key authenticated encryption

Example

```
#define MESSAGE (const unsigned char *) "test"
#define MESSAGE_LEN 4
#define CIPHERTEXT_LEN (crypto_box_MACBYTES + MESSAGE_LEN)

unsigned char alice_publickey[crypto_box_PUBLICKEYBYTES];
unsigned char alice_secretkey[crypto_box_SECRETKEYBYTES];
crypto_box_keypair(alice_publickey, alice_secretkey);

unsigned char bob_publickey[crypto_box_PUBLICKEYBYTES];
unsigned char bob_secretkey[crypto_box_SECRETKEYBYTES];
crypto_box_keypair(bob_publickey, bob_secretkey);

unsigned char nonce[crypto_box_NONCEBYTES];
unsigned char ciphertext[CIPHERTEXT_LEN];
randombytes_buf(nonce, sizeof nonce);
if (crypto_box_easy(ciphertext, MESSAGE, MESSAGE_LEN, nonce,
                   bob_publickey, alice_secretkey) != 0) {
    /* error */
}

unsigned char decrypted[MESSAGE_LEN];
if (crypto_box_open_easy(decrypted, ciphertext, CIPHERTEXT_LEN, nonce,
                        alice_publickey, bob_secretkey) != 0) {
    /* message for Bob pretending to be from Alice has been forged! */
}
```

Purpose

Using public-key authenticated encryption, Bob can encrypt a confidential message specifically for Alice, using Alice's public key.

Using Bob's public key, Alice can compute a shared secret key. Using Alice's public key and his secret key, Bob can compute the exact same shared secret key. That shared secret key can be used to verify that the encrypted message was not tampered with, before eventually decrypting it.

Alice only needs Bob's public key, the nonce and the ciphertext. Bob should never ever share his secret key, even with Alice.

And in order to send messages to Alice, Bob only needs Alice's public key. Alice should never ever share her secret key either, even with Bob.

Alice can reply to Bob using the same system, without having to generate a distinct key pair.

The nonce doesn't have to be confidential, but it should be used with just one invocation of `crypto_box_easy()` for a particular pair of public and secret keys.

One easy way to generate a nonce is to use `randombytes_buf()`, considering the size of the nonces the risk of any random collisions is negligible. For some applications, if you wish to use nonces to detect missing messages or to ignore replayed messages, it is also acceptable to use a simple incrementing counter as a nonce. A better alternative is to use the `crypto_secretstream()` API.

When doing so you must ensure that the same value can never be re-used (for example you may have multiple threads or even hosts generating messages using the same key pairs).

As stated above, senders can decrypt their own messages, and compute a valid authentication tag for any messages encrypted with a given shared secret key. This is generally not an issue for online protocols. If this is not acceptable, check out the Sealed Boxes section, as well as the Key Exchange section in this documentation.

Key pair generation

```
int crypto_box_keypair(unsigned char *pk, unsigned char *sk);
```

The `crypto_box_keypair()` function randomly generates a secret key and a corresponding public key. The public key is put into `pk` (`crypto_box_PUBLICKEYBYTES` bytes) and the secret key into `sk` (`crypto_box_SECRETKEYBYTES` bytes).

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

Using `crypto_box_seed_keypair()`, the key pair can also be deterministically derived from a single key `seed` (`crypto_box_SEEDBYTES` bytes).

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

In addition, `crypto_scalarmult_base()` can be used to compute the public key given a secret key previously generated with `crypto_box_keypair()`:

```
unsigned char pk[crypto_box_PUBKEYBYTES];
crypto_scalarmult_base(pk, sk);
```

Combined mode

In combined mode, the authentication tag and the encrypted message are stored together. This is usually what you want.

```
int crypto_box_easy(unsigned char *c, const unsigned char *m,
                   unsigned long long mlen, const unsigned char *n,
                   const unsigned char *pk, const unsigned char *sk);
```

The `crypto_box_easy()` function encrypts a message `m` whose length is `mlen` bytes, with a recipient's public key `pk`, a sender's secret key `sk` and a nonce `n`.

`n` should be `crypto_box_NONCEBYTES` bytes.

`c` should be at least `crypto_box_MACBYTES + mlen` bytes long.

This function writes the authentication tag, whose length is `crypto_box_MACBYTES` bytes, in `c`, immediately followed by the encrypted message, whose length is the same as the plaintext: `mlen`.

`c` and `m` can overlap, making in-place encryption possible. However do not forget that `crypto_box_MACBYTES` extra bytes are required to prepend the tag.

```
int crypto_box_open_easy(unsigned char *m, const unsigned char *c,
                       unsigned long long clen, const unsigned char *n,
                       const unsigned char *pk, const unsigned char *sk);
```

The `crypto_box_open_easy()` function verifies and decrypts a ciphertext produced by `crypto_box_easy()`.

`c` is a pointer to an authentication tag + encrypted message combination, as produced by `crypto_box_easy()`. `clen` is the length of this authentication tag + encrypted message combination. Put differently, `clen` is the number of bytes written by `crypto_box_easy()`, which is `crypto_box_MACBYTES` + the length of the message.

The nonce `n` has to match the nonce used to encrypt and authenticate the message.

`pk` is the public key of the sender that encrypted the message. `sk` is the secret key of the recipient that is willing to verify and decrypt it.

The function returns `-1` if the verification fails, and `0` on success. On success, the decrypted message is stored into `m`.

`m` and `c` can overlap, making in-place decryption possible.

Detached mode

Some applications may need to store the authentication tag and the encrypted message at different locations.

For this specific use case, "detached" variants of the functions above are available.

```
int crypto_box_detached(unsigned char *c, unsigned char *mac,
                        const unsigned char *m,
                        unsigned long long mlen,
                        const unsigned char *n,
                        const unsigned char *pk,
                        const unsigned char *sk);
```

This function encrypts a message `m` of length `mlen` with a nonce `n` and a secret key `sk` for a recipient whose public key is `pk`, and puts the encrypted message into `c`.

Exactly `mlen` bytes will be put into `c`, since this function does not prepend the authentication tag.

The tag, whose size is `crypto_box_MACBYTES` bytes, will be put into `mac`.

```
int crypto_box_open_detached(unsigned char *m,
                             const unsigned char *c,
                             const unsigned char *mac,
                             unsigned long long clen,
                             const unsigned char *n,
                             const unsigned char *pk,
                             const unsigned char *sk);
```

The `crypto_box_open_detached()` function verifies and decrypts an encrypted message `c` whose length is `clen` using the recipient's secret key `sk` and the sender's public key `pk`.

`clen` doesn't include the tag, so this length is the same as the plaintext.

The plaintext is put into `m` after verifying that `mac` is a valid authentication tag for this ciphertext, with the given nonce `n` and key `k`.

The function returns `-1` if the verification fails, or `0` on success.

Precalculation interface

Applications that send several messages to the same receiver or receive several messages from the same sender can gain speed by calculating the shared key only once, and reusing it in subsequent operations.

```
int crypto_box_beforenm(unsigned char *k, const unsigned char *pk,
                        const unsigned char *sk);
```

The `crypto_box_beforenm()` function computes a shared secret key given a public key `pk` and a secret key `sk`, and puts it into `k` (`crypto_box_BEFORENMBYTES` bytes).

```
int crypto_box_easy_afternm(unsigned char *c, const unsigned char *m,
                           unsigned long long mlen, const unsigned char *n,
                           const unsigned char *k);

int crypto_box_open_easy_afternm(unsigned char *m, const unsigned char *c,
                                unsigned long long clen, const unsigned char *n,
                                const unsigned char *k);

int crypto_box_detached_afternm(unsigned char *c, unsigned char *mac,
                                const unsigned char *m, unsigned long long mlen,
                                const unsigned char *n, const unsigned char *k);

int crypto_box_open_detached_afternm(unsigned char *m, const unsigned char *c,
                                     const unsigned char *mac,
                                     unsigned long long clen, const unsigned char *n,
                                     const unsigned char *k);
```

The `_afternm` variants of the previously described functions accept a precalculated shared secret key `k` instead of a key pair.

Like any secret key, a precalculated shared key should be wiped from memory (for example using `sodium_memzero()`) as soon as it is not needed any more.

`c` and `m` can overlap, making in-place encryption possible. However do not forget that `crypto_box_MACBYTES` extra bytes are required to prepend the tag.

Constants

- `crypto_box_PUBLICKEYBYTES`
- `crypto_box_SECRETKEYBYTES`
- `crypto_box_MACBYTES`
- `crypto_box_NONCEBYTES`

- `crypto_box_SEEDBYTES`
- `crypto_box_BEFORENMBYTES`

Algorithm details

- Key exchange: X25519
- Encryption: XSalsa20 stream cipher
- Authentication: Poly1305 MAC

Notes

The original NaCl `crypto_box` API is also supported, albeit not recommended.

`crypto_box()` takes a pointer to 32 bytes before the message, and stores the ciphertext 16 bytes after the destination pointer, the first 16 bytes being overwritten with zeros.

`crypto_box_open()` takes a pointer to 16 bytes before the ciphertext and stores the message 32 bytes after the destination pointer, overwriting the first 32 bytes with zeros.

The `_easy` and `_detached` APIs are faster and improve usability by not requiring padding, copying or tricky pointer arithmetic.

Public-key signatures

Example (combined mode)

```
#define MESSAGE (const unsigned char *) "test"
#define MESSAGE_LEN 4

unsigned char pk[crypto_sign_PUBLICKEYBYTES];
unsigned char sk[crypto_sign_SECRETKEYBYTES];
crypto_sign_keypair(pk, sk);

unsigned char signed_message[crypto_sign_BYTES + MESSAGE_LEN];
unsigned long long signed_message_len;

crypto_sign(signed_message, &signed_message_len,
            MESSAGE, MESSAGE_LEN, sk);

unsigned char unsigned_message[MESSAGE_LEN];
unsigned long long unsigned_message_len;
if (crypto_sign_open(unsigned_message, &unsigned_message_len,
                    signed_message, signed_message_len, pk) != 0) {
    /* Incorrect signature! */
}
```

Example (detached mode)

```
#define MESSAGE (const unsigned char *) "test"
#define MESSAGE_LEN 4

unsigned char pk[crypto_sign_PUBLICKEYBYTES];
unsigned char sk[crypto_sign_SECRETKEYBYTES];
crypto_sign_keypair(pk, sk);

unsigned char sig[crypto_sign_BYTES];

crypto_sign_detached(sig, NULL, MESSAGE, MESSAGE_LEN, sk);

if (crypto_sign_verify_detached(sig, MESSAGE, MESSAGE_LEN, pk) != 0) {
    /* Incorrect signature! */
}
```

Example (multi-part message)

```
#define MESSAGE_PART1 \
    ((const unsigned char *) "Arbitrary data to hash")
#define MESSAGE_PART1_LEN 22

#define MESSAGE_PART2 \
    ((const unsigned char *) "is longer than expected")
#define MESSAGE_PART2_LEN 23

unsigned char pk[crypto_sign_PUBLICKEYBYTES];
unsigned char sk[crypto_sign_SECRETKEYBYTES];
crypto_sign_keypair(pk, sk);

crypto_sign_state state;
unsigned char sig[crypto_sign_BYTES];

/* signature creation */

crypto_sign_init(&state)
crypto_sign_update(&state, MESSAGE_PART1, MESSAGE_PART1_LEN);
crypto_sign_update(&state, MESSAGE_PART2, MESSAGE_PART2_LEN);
crypto_sign_final_create(&state, sig, NULL, sk);

/* signature verification */

crypto_sign_init(&state)
crypto_sign_update(&state, MESSAGE_PART1, MESSAGE_PART1_LEN);
crypto_sign_update(&state, MESSAGE_PART2, MESSAGE_PART2_LEN);
if (crypto_sign_final_verify(&state, sig, pk) != 0) {
    /* message forged! */
}
```

Purpose

In this system, a signer generates a key pair:

- a secret key, that will be used to append a signature to any number of messages
- a public key, that anybody can use to verify that the signature appended to a message was actually issued by the creator of the public key.

Verifiers need to already know and ultimately trust a public key before messages signed using it can be verified.

Warning: this is different from authenticated encryption. Appending a signature does not change the representation of the message itself.

Key pair generation


```
int crypto_sign_keypair(unsigned char *pk, unsigned char *sk);
```

The `crypto_sign_keypair()` function randomly generates a secret key and a corresponding public key. The public key is put into `pk` (`crypto_sign_PUBLICKEYBYTES` bytes) and the secret key into `sk` (`crypto_sign_SECRETKEYBYTES` bytes).

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

Using `crypto_sign_seed_keypair()` , the key pair can also be deterministically derived from a single key `seed` (`crypto_sign_SEEDBYTES` bytes).

Combined mode

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

The `crypto_sign()` function prepends a signature to a message `m` whose length is `mlen` bytes, using the secret key `sk` .

The signed message, which includes the signature + a plain copy of the message, is put into `sm` , and is `crypto_sign_BYTES + mlen` bytes long.

If `sm_len` is not a `NULL` pointer, the actual length of the signed message is stored into `sm_len` .

```
int crypto_sign_open(unsigned char *m, unsigned long long *m_len_p,
                    const unsigned char *sm, unsigned long long sm_len,
                    const unsigned char *pk);
```

The `crypto_sign_open()` function checks that the signed message `sm` whose length is `sm_len` bytes has a valid signature for the public key `pk` .

If the signature is doesn't appear to be valid, the function returns `-1` .

On success, it puts the message with the signature removed into `m` , stores its length into `m_len` if `m_len` is not a `NULL` pointer, and returns `0` .

Detached mode

In detached mode, the signature is stored without attaching a copy of the original message to it.

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

The `crypto_sign_detached()` function signs the message `m` whose length is `mlen` bytes, using the secret key `sk`, and puts the signature into `sig`, which can be up to `crypto_sign_BYTES` bytes long.

The actual length of the signature is put into `siglen` if `siglen` is not `NULL`.

It is safe to ignore `siglen` and always consider a signature as `crypto_sign_BYTES` bytes long: shorter signatures will be transparently padded with zeros if necessary.

```
int crypto_sign_verify_detached(const unsigned char *sig,
                               const unsigned char *m,
                               unsigned long long mlen,
                               const unsigned char *pk);
```

The `crypto_sign_verify_detached()` function verifies that `sig` is a valid signature for the message `m` whose length is `mlen` bytes, using the signer's public key `pk`.

It returns `-1` if the signature fails verification, or `0` on success.

Multi-part messages

If the message doesn't fit in memory, it can be provided as a sequence of arbitrarily-sized chunks.

This will use the Ed25519ph signature system, that pre-hashes the message. In other words, what gets signed is not the message itself, but its image through a hash function.

If the message *can* fit in memory and can be supplied as a single chunk, the single-part API should be preferred.

Note: `Ed25519ph(m)` is intentionally not equivalent to `Ed25519(SHA512(m))`.

If, for some reason, you need to prehash the message yourself, use the multi-part `crypto_generichash_*`() APIs and sign the 512 bit output.

```
int crypto_sign_init(crypto_sign_state *state);
```

The `crypto_sign_init()` function initializes the state `state`. This function must be called before the first `crypto_sign_update()` call.

```
int crypto_sign_update(crypto_sign_state *state,
                      const unsigned char *m, unsigned long long mlen);
```

Add a new chunk `m` of length `mlen` bytes to the message that will eventually be signed.

After all parts have been supplied, one of the following functions can be called:

```
int crypto_sign_final_create(crypto_sign_state *state, unsigned char *sig,
                           unsigned long long *siglen_p,
                           const unsigned char *sk);
```

The `crypto_sign_final_create()` function computes a signature for the previously supplied message, using the secret key `sk` and puts it into `sig`.

If `siglen_p` is not `NULL`, the length of the signature is stored at this address.

It is safe to ignore `siglen` and always consider a signature as `crypto_sign_BYTES` bytes long: shorter signatures will be transparently padded with zeros if necessary.

```
int crypto_sign_final_verify(crypto_sign_state *state, unsigned char *sig,
                            const unsigned char *pk);
```

The `crypto_sign_final_verify()` function verifies that `sig` is a valid signature for the message whose content has been previously supplied using `crypto_update()`, using the public key `pk`.

Extracting the seed and the public key from the secret key

The secret key actually includes the seed (either a random seed or the one given to `crypto_sign_seed_keypair()`) as well as the public key.

While the public key can always be derived from the seed, the precomputation saves a significant amount of CPU cycles when signing.

If required, Sodium provides two functions to extract the seed and the public key from the secret key:

```
int crypto_sign_ed25519_sk_to_seed(unsigned char *seed,
                                   const unsigned char *sk);

int crypto_sign_ed25519_sk_to_pk(unsigned char *pk, const unsigned char *sk);
```

The `crypto_sign_ed25519_sk_to_seed()` function extracts the seed from the secret key `sk` and copies it into `seed` (`crypto_sign_SEEDBYTES` bytes).

The `crypto_sign_ed25519_sk_to_pk()` function extracts the public key from the secret key `sk` and copies it into `pk` (`crypto_sign_PUBLICKEYBYTES` bytes).

Data structures

- `crypto_sign_state` , whose size can be retrieved using `crypto_sign_statebytes()`

Constants

- `crypto_sign_PUBLICKEYBYTES`
- `crypto_sign_SECRETKEYBYTES`
- `crypto_sign_BYTES`
- `crypto_sign_SEEDBYTES`

Algorithm details

- Single-part signature: Ed25519
- Multi-part signature: Ed25519ph

References

- [Edwards-Curve Digital Signature Algorithm \(EdDSA\)](#)

Notes

`crypto_sign_verify()` and `crypto_sign_verify_detached()` are only designed to verify signatures computed using `crypto_sign()` and `crypto_sign_detached()` .

The original NaCl `crypto_sign_open()` implementation overwrote 64 bytes after the message. The libsodium implementation doesn't write past the end of the message.

Ed25519ph (used by the multi-part API) was implemented in libsodium 1.0.12.

The Ed25519 system was designed to compute deterministic signatures.

Non-deterministic (but also non-standard) signatures can be produced by compiling libsodium with the `ED25519_NONDETERMINISTIC` macro defined.

Sealed boxes

Example

```
#define MESSAGE (const unsigned char *) "Message"
#define MESSAGE_LEN 7
#define CIPHERTEXT_LEN (crypto_box_SEALBYTES + MESSAGE_LEN)

/* Recipient creates a long-term key pair */
unsigned char recipient_pk[crypto_box_PUBLICKEYBYTES];
unsigned char recipient_sk[crypto_box_SECRETKEYBYTES];
crypto_box_keypair(recipient_pk, recipient_sk);

/* Anonymous sender encrypts a message using an ephemeral key pair
 * and the recipient's public key */
unsigned char ciphertext[CIPHERTEXT_LEN];
crypto_box_seal(ciphertext, MESSAGE, MESSAGE_LEN, recipient_pk);

/* Recipient decrypts the ciphertext */
unsigned char decrypted[MESSAGE_LEN];
if (crypto_box_seal_open(decrypted, ciphertext, CIPHERTEXT_LEN,
                        recipient_pk, recipient_sk) != 0) {
    /* message corrupted or not intended for this recipient */
}
```

Purpose

Sealed boxes are designed to anonymously send messages to a recipient given its public key.

Only the recipient can decrypt these messages, using its private key. While the recipient can verify the integrity of the message, it cannot verify the identity of the sender.

A message is encrypted using an ephemeral key pair, whose secret part is destroyed right after the encryption process.

Without knowing the secret key used for a given message, the sender cannot decrypt its own message later. And without additional data, a message cannot be correlated with the identity of its sender.

Usage

```
int crypto_box_seal(unsigned char *c, const unsigned char *m,
                    unsigned long long mlen, const unsigned char *pk);
```

The `crypto_box_seal()` function encrypts a message `m` of length `mlen` for a recipient whose public key is `pk`. It puts the ciphertext whose length is `crypto_box_SEALBYTES + mlen` into `c`.

The function creates a new key pair for each message, and attaches the public key to the ciphertext. The secret key is overwritten and is not accessible after this function returns.

```
int crypto_box_seal_open(unsigned char *m, const unsigned char *c,
                         unsigned long long clen,
                         const unsigned char *pk, const unsigned char *sk);
```

The `crypto_box_seal_open()` function decrypts the ciphertext `c` whose length is `clen`, using the key pair (`pk`, `sk`), and puts the decrypted message into `m` (`clen - crypto_box_SEALBYTES` bytes).

Key pairs are compatible with other `crypto_box_*` operations and can be created using `crypto_box_keypair()` or `crypto_box_seed_keypair()`.

This function doesn't require passing the public key of the sender, as the ciphertext already includes this information.

Constants

- `crypto_box_SEALBYTES`

Algorithm details

Sealed boxes leverage the `crypto_box` construction (X25519, XSalsa20-Poly1305).

The format of a sealed box is

```
ephemeral_pk || box(m, recipient_pk, ephemeral_sk, nonce=blake2b(ephemeral_pk || recipient_pk))
```

Hashing

(this documentation is a work in progress. Feel free to contribute a nice intro to hash functions!)

Generic hashing

Single-part example without a key

```
#define MESSAGE ((const unsigned char *) "Arbitrary data to hash")
#define MESSAGE_LEN 22

unsigned char hash[crypto_generichash_BYTES];

crypto_generichash(hash, sizeof hash,
                   MESSAGE, MESSAGE_LEN,
                   NULL, 0);
```

Single-part example with a key

```
#define MESSAGE ((const unsigned char *) "Arbitrary data to hash")
#define MESSAGE_LEN 22

unsigned char hash[crypto_generichash_BYTES];
unsigned char key[crypto_generichash_KEYBYTES];

randombytes_buf(key, sizeof key);

crypto_generichash(hash, sizeof hash,
                   MESSAGE, MESSAGE_LEN,
                   key, sizeof key);
```

Multi-part example with a key

```
#define MESSAGE_PART1 \  
    ((const unsigned char *) "Arbitrary data to hash")  
#define MESSAGE_PART1_LEN 22  
  
#define MESSAGE_PART2 \  
    ((const unsigned char *) "is longer than expected")  
#define MESSAGE_PART2_LEN 23  
  
unsigned char hash[crypto_generichash_BYTES];  
unsigned char key[crypto_generichash_KEYBYTES];  
crypto_generichash_state state;  
  
randombytes_buf(key, sizeof key);  
  
crypto_generichash_init(&state, key, sizeof key, sizeof hash);  
  
crypto_generichash_update(&state, MESSAGE_PART1, MESSAGE_PART1_LEN);  
crypto_generichash_update(&state, MESSAGE_PART2, MESSAGE_PART2_LEN);  
  
crypto_generichash_final(&state, hash, sizeof hash);
```

Purpose

This API computes a fixed-length fingerprint for an arbitrary long message.

Sample use cases:

- File integrity checking
- Creating unique identifiers to index arbitrary long data

Usage

```
int crypto_generichash(unsigned char *out, size_t outlen,  
    const unsigned char *in, unsigned long long inlen,  
    const unsigned char *key, size_t keylen);
```

The `crypto_generichash()` function puts a fingerprint of the message `in` whose length is `inlen` bytes into `out`. The output size can be chosen by the application.

The minimum recommended output size is `crypto_generichash_BYTES`. This size makes it practically impossible for two messages to produce the same fingerprint.

But for specific use cases, the size can be any value between

`crypto_generichash_BYTES_MIN` (included) and `crypto_generichash_BYTES_MAX` (included).

`key` can be `NULL` and `keylen` can be `0`. In this case, a message will always have the same fingerprint, similar to the `MD5` or `SHA-1` functions for which `crypto_generichash()` is a faster and more secure alternative.

But a key can also be specified. A message will always have the same fingerprint for a given key, but different keys used to hash the same message are very likely to produce distinct fingerprints.

In particular, the key can be used to make sure that different applications generate different fingerprints even if they process the same data.

The recommended key size is `crypto_generichash_KEYBYTES` bytes.

However, the key size can be any value between `crypto_generichash_KEYBYTES_MIN` (included) and `crypto_generichash_KEYBYTES_MAX` (included).

```
int crypto_generichash_init(crypto_generichash_state *state,
                           const unsigned char *key,
                           const size_t keylen, const size_t outlen);

int crypto_generichash_update(crypto_generichash_state *state,
                             const unsigned char *in,
                             unsigned long long inlen);

int crypto_generichash_final(crypto_generichash_state *state,
                            unsigned char *out, const size_t outlen);
```

The message doesn't have to be provided as a single chunk. The `generichash` operation also supports a streaming API.

The `crypto_generichash_init()` function initializes a state `state` with a key `key` (that can be `NULL`) of length `keylen` bytes, in order to eventually produce `outlen` bytes of output.

Each chunk of the complete message can then be sequentially processed by calling `crypto_generichash_update()`, providing the previously initialized state `state`, a pointer to the chunk `in` and the length of the chunk in bytes, `inlen`.

The `crypto_generichash_final()` function completes the operation and puts the final fingerprint into `out` as `outlen` bytes.

After `crypto_generichash_final()` returns, the state should not be used any more, unless it is reinitialized using `crypto_generichash_init()`.

This alternative API is especially useful to process very large files and data streams.

```
void crypto_generichash_keygen(unsigned char k[crypto_generichash_KEYBYTES]);
```

The `crypto_generichash_keygen()` function creates a key `k` of the recommended length `crypto_generichash_KEYBYTES`.

State structure size

The `crypto_generichash_state` structure length is either 357 or 361 bytes. 64-bytes alignment is required. For dynamically allocated states, `crypto_generichash_statebytes()` returns the rounded up structure size, and should be preferred to `sizeof()`.

```
state = sodium_malloc(crypto_generichash_statebytes());
```

Constants

- `crypto_generichash_BYTES`
- `crypto_generichash_BYTES_MIN`
- `crypto_generichash_BYTES_MAX`
- `crypto_generichash_KEYBYTES`
- `crypto_generichash_KEYBYTES_MIN`
- `crypto_generichash_KEYBYTES_MAX`

Data types

- `crypto_generichash_state`

Algorithm details

BLAKE2b

Notes

The `crypto_generichash_*` function set is implemented using BLAKE2b, a simple, standardized ([RFC 7693](#)) secure hash function that is as strong as SHA-3 but faster than SHA-1 and MD5.

Unlike MD5, SHA-1 and SHA-256, this function is safe against hash length extension attacks.

BLAKE2b's salt and personalisation parameters are accessible through the lower-level functions whose prototypes are defined in `crypto_generichash_blake2b.h`.

BLAKE2b is not suitable for hashing passwords. For this purpose, use the `crypto_pwhash` API documented in the Password Hashing section.

Short-input hashing

Example

```
#define SHORT_DATA ((const unsigned char *) "Sparkling water")
#define SHORT_DATA_LEN 15

unsigned char hash[crypto_shorthash_BYTES];
unsigned char key[crypto_shorthash_KEYBYTES];

crypto_shorthash_keygen(key);
crypto_shorthash(hash, SHORT_DATA, SHORT_DATA_LEN, key);
```

Purpose

Many applications and programming language implementations were recently found to be vulnerable to denial-of-service attacks when a hash function with weak security guarantees, such as Murmurhash 3, was used to construct a hash table.

In order to address this, Sodium provides the `crypto_shorthash()` function, which outputs short but unpredictable (without knowing the secret key) values suitable for picking a list in a hash table for a given key.

This function is optimized for short inputs.

The output of this function is only 64 bits. Therefore, it should *not* be considered collision-resistant.

Use cases:

- Hash tables
- Probabilistic data structures such as Bloom filters
- Integrity checking in interactive protocols

Usage

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

Compute a fixed-size (`crypto_shorthash_BYTES` bytes) fingerprint for the message `in` whose length is `inlen` bytes, using the key `k` .

The `k` is `crypto_shorthash_KEYBYTES` bytes and can be created using `crypto_shorthash_keygen()` .

The same message hashed with the same key will always produce the same output.

Constants

- `crypto_shorthash_BYTES`
- `crypto_shorthash_KEYBYTES`

Algorithm details

SipHash-2-4

Notes

- The key has to remain secret. This function will not provide any mitigations against DoS attacks if the key is known from attackers.
- When building hash tables, it is recommended to use a prime number for the table size. This ensures that all bits from the output of the hash function are being used. Mapping the range of the hash function to `[0..N)` can be done efficiently [without modulo reduction](#).
- libsodium >= 1.0.12 also implements a variant of SipHash with the same key size but a 128-bit output, accessible as `crypto_shorthash_siphashx24()` .

Password hashing

Secret keys used to encrypt or sign confidential data have to be chosen from a very large keyspace.

However, passwords are usually short, human-generated strings, making dictionary attacks practical.

Password hashing functions derive a secret key of any size from a password and a salt.

- The generated key has the size defined by the application, no matter what the password length is.
- The same password hashed with same parameters will always produce the same output.
- The same password hashed with different salts will produce different outputs.
- The function deriving a key from a password and a salt is CPU intensive and intentionally requires a fair amount of memory. Therefore, it mitigates brute-force attacks by requiring a significant effort to verify each password.

Common use cases:

- Password storage, or rather: storing what it takes to verify a password without having to store the actual password.
- Deriving a secret key from a password, for example for disk encryption.

Sodium's high-level `crypto_pwhash_*` API currently leverages the Argon2id function on all platforms. This can change at any point in time, but it is guaranteed that a given version of libsodium can verify all hashes produced by all previous versions, from any platform. Applications don't have to worry about backward compatibility.

The more specific `crypto_pwhash_scryptsalsa208sha256_*` API uses the more conservative and widely deployed Scrypt function.

Argon2

Argon2 is optimized for the x86 architecture and exploits the cache and memory organization of the recent Intel and AMD processors. But its implementation remains portable and fast on other architectures, with the exception of JavaScript.

Argon2 has three variants: Argon2d, Argon2i and Argon2id. Libsodium supports Argon2i and Argon2id.

Script

Script was also designed to make it costly to perform large-scale custom hardware attacks by requiring large amounts of memory.

Even though its memory hardness can be significantly reduced at the cost of extra computations, this function remains an excellent choice today, provided that its parameters are properly chosen.

Server relief

If multiple clients can simultaneously log in on a shared server, the memory and computation requirements can exhaust the server's resources.

In order to mitigate this, passwords can be pre-hashed on the client (e.g. using `libsodium.js` in a web application):

- On user account creation, the server sends a random seed to the client. The client computes `ph = password_hash(password, seed)` and sends `ph` to the server. `password_hash` is a password hashing function tuned for the maximum memory and CPU usage the client can handle. The server stores the seed and `password_hash'(ph, seed)` for this user account. `password_hash'` is a password hashing function, whose parameters can be tuned for low memory and CPU usage.
- On a login attempt, the server sends the seed, or, for a nonexistent user, a pseudorandom seed that has to always be the same for a given user name (for example using `crypto_generichash()`, with a key, and the user name as the message). The client computes `ph = password_hash(password, seed)` and sends it to the server. The server computes `password_hash'(ph, seed)` and compares it against what was stored in the database.

The default password-hashing function

Sodium provides an API that can be used both for key derivation using a low-entropy input, and for password storage.

Example 1: key derivation

```
#define PASSWORD "Correct Horse Battery Staple"
#define KEY_LEN crypto_box_SEEDBYTES

unsigned char salt[crypto_pwhash_SALTBYTES];
unsigned char key[KEY_LEN];

randombytes_buf(salt, sizeof salt);

if (crypto_pwhash
    (key, sizeof key, PASSWORD, strlen(PASSWORD), salt,
     crypto_pwhash_OPSLIMIT_INTERACTIVE, crypto_pwhash_MEMLIMIT_INTERACTIVE,
     crypto_pwhash_ALG_DEFAULT) != 0) {
    /* out of memory */
}
```

Example 2: password storage

```
#define PASSWORD "Correct Horse Battery Staple"

char hashed_password[crypto_pwhash_STRBYTES];

if (crypto_pwhash_str
    (hashed_password, PASSWORD, strlen(PASSWORD),
     crypto_pwhash_OPSLIMIT_SENSITIVE, crypto_pwhash_MEMLIMIT_SENSITIVE) != 0) {
    /* out of memory */
}

if (crypto_pwhash_str_verify
    (hashed_password, PASSWORD, strlen(PASSWORD)) != 0) {
    /* wrong password */
}
```

Key derivation

```
int crypto_pwhash(unsigned char * const out,
                  unsigned long long outlen,
                  const char * const passwd,
                  unsigned long long passwdlen,
                  const unsigned char * const salt,
                  unsigned long long opslimit,
                  size_t memlimit, int alg);
```

The `crypto_pwhash()` function derives an `outlen` bytes long key from a password `passwd` whose length is `passwdlen` and a salt `salt` whose fixed length is `crypto_pwhash_SALTBYTES` bytes. `passwdlen` should be at least `crypto_pwhash_PASSWD_MIN` and `crypto_pwhash_PASSWD_MAX`. `outlen` should be at least `crypto_pwhash_BYTES_MIN` = 16 (128 bits) and at most `crypto_pwhash_BYTES_MAX`.

The computed key is stored into `out`.

`opslimit` represents a maximum amount of computations to perform. Raising this number will make the function require more CPU cycles to compute a key. This number must be between `crypto_pwhash_OPSLIMIT_MIN` and `crypto_pwhash_OPSLIMIT_MAX`.

`memlimit` is the maximum amount of RAM that the function will use, in bytes. This number must be between `crypto_pwhash_MEMLIMIT_MIN` and `crypto_pwhash_MEMLIMIT_MAX`.

`alg` is an identifier for the algorithm to use, and should be set to one of the following values:

- `crypto_pwhash_ALG_DEFAULT` : the currently recommended algorithm, which can change from one version of libsodium to another.
- `crypto_pwhash_ALG_ARGON2I13` : version 1.3 of the Argon2i algorithm.
- `crypto_pwhash_ALG_ARGON2ID13` : version 1.3 of the Argon2id algorithm, available since libsodium 1.0.13.

For interactive, online operations, `crypto_pwhash_OPSLIMIT_INTERACTIVE` and `crypto_pwhash_MEMLIMIT_INTERACTIVE` provide base line for these two parameters. This currently requires 64 MiB of dedicated RAM. Higher values may improve security (see below).

Alternatively, `crypto_pwhash_OPSLIMIT_MODERATE` and `crypto_pwhash_MEMLIMIT_MODERATE` can be used. This requires 256 MiB of dedicated RAM, and takes about 0.7 seconds on a 2.8 Ghz Core i7 CPU.

For highly sensitive data and non-interactive operations, `crypto_pwhash_OPSLIMIT_SENSITIVE` and `crypto_pwhash_MEMLIMIT_SENSITIVE` can be used. With these parameters, deriving a key takes about 3.5 seconds on a 2.8 Ghz Core i7 CPU and requires 1024 MiB of dedicated RAM.

The `salt` should be unpredictable. `randombytes_buf()` is the easiest way to fill the `crypto_pwhash_SALTBYTES` bytes of the salt.

Keep in mind that in order to produce the same key from the same password, the same algorithm, the same salt, and the same values for `opslimit` and `memlimit` have to be used. Therefore, these parameters have to be stored for each user.

The function returns `0` on success, and `-1` if the computation didn't complete, usually because the operating system refused to allocate the amount of requested memory.

Password storage

```
int crypto_pwhash_str(char out[crypto_pwhash_STRBYTES],
                     const char * const passwd,
                     unsigned long long passwdlen,
                     unsigned long long opslimit,
                     size_t memlimit);
```

The `crypto_pwhash_str()` function puts an ASCII encoded string into `out`, which includes:

- the result of a memory-hard, CPU-intensive hash function applied to the password `passwd` of length `passwdlen`
- the automatically generated salt used for the previous computation
- the other parameters required to verify the password, including the algorithm identifier, its version, `opslimit` and `memlimit`.

`out` must be large enough to hold `crypto_pwhash_STRBYTES` bytes, but the actual output string may be shorter.

The output string is zero-terminated, includes only ASCII characters and can be safely stored into SQL databases and other data stores. No extra information has to be stored in order to verify the password.

The function returns `0` on success and `-1` if it didn't complete successfully.

```
int crypto_pwhash_str_verify(const char str[crypto_pwhash_STRBYTES],
                             const char * const passwd,
                             unsigned long long passwdlen);
```

This function verifies that `str` is a valid password verification string (as generated by `crypto_pwhash_str()`) for `passwd` whose length is `passwdlen`.

`str` has to be zero-terminated.

It returns `0` if the verification succeeds, and `-1` on error.

```
int crypto_pwhash_str_needs_rehash(const char str[crypto_pwhash_STRBYTES],
                                   unsigned long long opslimit, size_t memlimit);
```

Check if a password verification string `str` matches the parameters `opslimit` and `memlimit`, and the current default algorithm.

The function returns `1` if the string appears to be correct, but doesn't match the given parameters. In that situation, applications may want to compute a new hash using the current parameters the next time the user logs in.

The function returns `0` if the parameters already match the given ones.

It returns `-1` on error. If it happens, applications may want to compute a correct hash the next time the user logs in.

Guidelines for choosing the parameters

Start by determining how much memory the function can use. What will be the highest number of threads/processes evaluating the function simultaneously (ideally, no more than 1 per CPU core)? How much physical memory is guaranteed to be available?

Set `memlimit` to the amount of memory you want to reserve for password hashing.

Then, set `opslimit` to `3` and measure the time it takes to hash a password.

If this it is way too long for your application, reduce `memlimit`, but keep `opslimit` set to `3`.

If the function is so fast that you can afford it to be more computationally intensive without any usability issues, increase `opslimit`.

For online use (e.g. login in on a website), a 1 second computation is likely to be the acceptable maximum.

For interactive use (e.g. a desktop application), a 5 second pause after having entered a password is acceptable if the password doesn't need to be entered more than once per session.

For non-interactive use and infrequent use (e.g. restoring an encrypted backup), an even slower computation can be an option.

But the best defense against brute-force password cracking remains using strong passwords. Libraries such as [passwdqc](#) can help enforce this.

Constants

- `crypto_pwhash_ALG_ARGON2I13`
- `crypto_pwhash_ALG_ARGON2ID13`
- `crypto_pwhash_ALG_DEFAULT`
- `crypto_pwhash_BYTES_MAX`
- `crypto_pwhash_BYTES_MIN`
- `crypto_pwhash_MEMLIMIT_INTERACTIVE`
- `crypto_pwhash_MEMLIMIT_MAX`
- `crypto_pwhash_MEMLIMIT_MIN`
- `crypto_pwhash_MEMLIMIT_MODERATE`
- `crypto_pwhash_MEMLIMIT_SENSITIVE`
- `crypto_pwhash_OPSLIMIT_INTERACTIVE`
- `crypto_pwhash_OPSLIMIT_MAX`
- `crypto_pwhash_OPSLIMIT_MIN`
- `crypto_pwhash_OPSLIMIT_MODERATE`
- `crypto_pwhash_OPSLIMIT_SENSITIVE`
- `crypto_pwhash_PASSWD_MAX`
- `crypto_pwhash_PASSWD_MIN`
- `crypto_pwhash_SALTBYTES`
- `crypto_pwhash_STRBYTES`
- `crypto_pwhash_STRPREFIX`

Notes

`opslimit` , the number of passes, has to be at least `3` when using Argon2i. `crypto_pwhash()` and `crypto_pwhash_str()` will fail with a `-1` return code for lower values.

There is no "insecure" value for `memlimit` , though the more memory the better.

Do not forget to initialize the library with `sodium_init()` . `crypto_pwhash_*` will still work without doing so, but possibly way slower.

Do not use constants (including `crypto_pwhash_OPSLIMIT_*` and `crypto_pwhash_MEMLIMIT_*`) in order to verify a password or produce a deterministic output. Save the parameters (including the algorithm identifier) along with the hash instead.

For password verification, the recommended interface is `crypto_pwhash_str()` and `crypto_pwhash_str_verify()` . The string produced by `crypto_pwhash_str()` already includes an algorithm identifier, as well as all the parameters (including the automatically generated

salt) that have been used to hash the password. Subsequently, `crypto_pwhash_str_verify()` automatically decodes these parameters.

By doing so, passwords can be rehashed using different parameters if required later on.

Cleartext passwords should not stay in memory longer than needed.

It is highly recommended to use `sodium_mlock()` to lock memory regions storing cleartext passwords, and to call `sodium_munlock()` right after `crypto_pwhash_str()` and `crypto_pwhash_str_verify()` return.

`sodium_munlock()` overwrites the region with zeros before unlocking it, so it must not be done before calling this function (otherwise zeroes, instead of the password, would be hashed).

Since version 1.0.15, libsodium's default algorithm is Argon2id.

Algorithm details

- [Argon2 v1.3](#)

Key derivation

Deriving a key from a password

Secret keys used to encrypt or sign confidential data have to be chosen from a very large keyspace. However, passwords are usually short, human-generated strings, making dictionary attacks practical.

The `pwhash` operation derives a secret key of any size from a password and a salt.

Refer to the [Password hashing](#) section for more information and code examples.

Deriving keys from a single high-entropy key

Multiple secret subkeys can be derived from a single master key.

Given the master key and a key identifier, a subkey can be deterministically computed. However, given a subkey, an attacker cannot compute the master key nor any other subkeys.

Key derivation with libsodium >= 1.0.12

Recent versions of the library have a dedicated API for key derivation.

The `crypto_kdf` API can derive up to 2^{64} keys from a single master key and context, and individual subkeys can have an arbitrary length between 128 (16 bytes) and 512 bits (64 bytes).

Example:


```
#define CONTEXT "Examples"

uint8_t master_key[crypto_kdf_KEYBYTES];
uint8_t subkey1[32];
uint8_t subkey2[32];
uint8_t subkey3[64];

crypto_kdf_keygen(master_key);

crypto_kdf_derive_from_key(subkey1, sizeof subkey1, 1, CONTEXT, master_key);
crypto_kdf_derive_from_key(subkey2, sizeof subkey2, 2, CONTEXT, master_key);
crypto_kdf_derive_from_key(subkey3, sizeof subkey3, 3, CONTEXT, master_key);
```

Usage:

```
void crypto_kdf_keygen(uint8_t key[crypto_kdf_KEYBYTES]);
```

The `crypto_kdf_keygen()` function creates a master key.

```
int crypto_kdf_derive_from_key(unsigned char *subkey, size_t subkey_len,
                                uint64_t subkey_id,
                                const char ctx[crypto_kdf_CONTEXTBYTES],
                                const unsigned char key[crypto_kdf_KEYBYTES]);
```

The `crypto_kdf_derive_from_key()` function derives a `subkey_id`-th subkey `subkey` of length `subkey_len` bytes using the master key `key` and the context `ctx`.

`subkey_id` can be any value up to $(2^{64})-1$.

`subkey_len` has to be between `crypto_kdf_BYTES_MIN` (inclusive) and `crypto_kdf_BYTES_MAX` (inclusive).

Similar to a type, the context `ctx` is a 8 characters string describing what the key is going to be used for.

Its purpose is to mitigate accidental bugs by separating domains. The same function used with the same key but in two distinct contexts is likely to generate two different outputs.

Contexts don't have to be secret and can have a low entropy.

Examples of contexts include `UserName`, `__auth__`, `pictures` and `userdata`.

If more convenient, it is also fine to use a single global context for a whole application. This will still prevent the same keys from being mistakenly used by another application.

Constants:

- `crypto_kdf_PRIMITIVE`
- `crypto_kdf_BYTES_MIN`
- `crypto_kdf_BYTES_MAX`
- `crypto_kdf_CONTEXTBYTES`
- `crypto_kdf_KEYBYTES`

Algorithm details:

```
BLAKE2B-subkeylen(key=key, message={}, salt=subkey_id || {0}, personal=ctx || {0})
```

Key derivation with libsodium < 1.0.12

On older versions of the library, the BLAKE2 function can be used directly:

```
const unsigned char appid[crypto_generichash_blake2b_PERSONALBYTES] = {
    'A', ' ', 'S', 'i', 'm', 'p', 'l', 'e', ' ', 'E', 'x', 'a', 'm', 'p', 'l', 'e'
};
unsigned char keyid[crypto_generichash_blake2b_SALTBYTES] = {0};
unsigned char masterkey[64];
unsigned char subkey1[16];
unsigned char subkey2[32];

/* Generate a master key */
randombytes_buf(masterkey, sizeof masterkey);

/* Derive a first subkey (id=0) */
crypto_generichash_blake2b_salt_personal(subkey1, sizeof subkey1,
                                         NULL, 0,
                                         masterkey, sizeof masterkey,
                                         keyid, appid);

/* Derive a second subkey (id=1) */
sodium_increment(keyid, sizeof keyid);
crypto_generichash_blake2b_salt_personal(subkey2, sizeof subkey2,
                                         NULL, 0,
                                         masterkey, sizeof masterkey,
                                         keyid, appid);
```

The `crypto_generichash_blake2b_salt_personal()` function can be used to derive a subkey of any size from a key of any size, as long as these key sizes are in the 128 to 512 bits interval.

In this example, two subkeys are derived from a single key. These subkeys have different sizes (128 and 256 bits), and are derived from a master key of yet another size (512 bits).

The personalization parameter (`appid`) is a 16-bytes value that doesn't have to be secret. It can be used so that the same (`masterkey`, `keyid`) tuple will produce different output in different applications. It is not required, however: a `NULL` pointer can be passed instead in order to use the default constant.

The salt (`keyid`) doesn't have to be secret either. This is a 16-bytes identifier, that can be a simple counter, and is used to derive more than one key out of a single master key.

Nonce extension

Unlike XSalsa20 (used by `crypto_box_*` and `crypto_secretbox_*`) and XChaCha20, ciphers such as AES-GCM and ChaCha20 require a nonce too short to be chosen randomly (64 or 96 bits). With 96 bits random nonces, 2^{32} encryptions is the limit before the probability of duplicate nonces becomes too high.

Using a counter instead of random nonces prevents this. However, keeping a state is not always an option, especially with offline protocols.

As an alternative, the nonce can be extended: a key and a part of a long nonce are used as inputs to a pseudorandom function to compute a new key. This subkey and the remaining bits of the long nonce can then be used as parameters for the cipher.

For example, this allows using a 192-bits nonce with a cipher requiring a 64-bits nonce:

```
k = <key>
n = <192-bit nonce>
k' = PRF(k, n[0..127])
c = E(k', n[128..191], m)
```

Since version 1.0.9, Sodium provides the `crypto_core_hchacha20()` function, which can be used as a PRF for that purpose:

```
int crypto_core_hchacha20(unsigned char *out, const unsigned char *in,
                          const unsigned char *k, const unsigned char *c);
```

This function accepts a 32 bytes (`crypto_core_hchacha20_KEYBYTES`) secret key `k` as well as a 16 bytes (`crypto_core_hchacha20_INPUTBYTES`) input `in` , and outputs a 32 bytes (`crypto_core_hchacha20_OUTPUTBYTES`) value indistinguishable from random data without knowing `k` .

Optionally, a 16-bytes (`crypto_core_hchacha20_CONSTBYTES`) constant `c` can be specified to personalize the function to an application. `c` can be left to `NULL` in order to use the default constant.

The following code snippet case thus be used to construct a ChaCha20-Poly1305 variant with a 192-bits nonce (XChaCha20) on libsodium < 1.0.12 (versions >= 1.0.12 already include this construction).

```
#define MESSAGE (const unsigned char *) "message"
#define MESSAGE_LEN 7

unsigned char c[crypto_aead_chacha20poly1305_ABYTES + MESSAGE_LEN];
unsigned char k[crypto_core_hchacha20_KEYBYTES];
unsigned char k2[crypto_core_hchacha20_OUTPUTBYTES];
unsigned char n[crypto_core_hchacha20_INPUTBYTES +
                crypto_aead_chacha20poly1305_NPUBBYTES];

randombytes_buf(k, sizeof k);
randombytes_buf(n, sizeof n); /* 192-bits nonce */

crypto_core_hchacha20(k2, n, k, NULL);

assert(crypto_aead_chacha20poly1305_KEYBYTES <= sizeof k2);
assert(crypto_aead_chacha20poly1305_NPUBBYTES ==
       (sizeof n) - crypto_core_hchacha20_INPUTBYTES);

crypto_aead_chacha20poly1305_encrypt(c, NULL, MESSAGE, MESSAGE_LEN,
                                     NULL, 0, NULL,
                                     n + crypto_core_hchacha20_INPUTBYTES,
                                     k2);
```

Key exchange

Example (client-side)

```
unsigned char client_pk[crypto_kx_PUBKEYBYTES], client_sk[crypto_kx_SECRETKEYBYTES]
;
unsigned char client_rx[crypto_kx_SESSIONKEYBYTES], client_tx[crypto_kx_SESSIONKEYBYTES];

/* Generate the client's key pair */
crypto_kx_keypair(client_pk, client_sk);

/* Prerequisite after this point: the server's public key must be known by the client
*/

/* Compute two shared keys using the server's public key and the client's secret key.
   client_rx will be used by the client to receive data from the server,
   client_tx will be used by the client to send data to the server. */
if (crypto_kx_client_session_keys(client_rx, client_tx,
                                  client_pk, client_sk, server_pk) != 0) {
    /* Suspicious server public key, bail out */
}
```

Example (server-side)

```
unsigned char server_pk[crypto_kx_PUBKEYBYTES], server_sk[crypto_kx_SECRETKEYBYTES]
;
unsigned char server_rx[crypto_kx_SESSIONKEYBYTES], server_tx[crypto_kx_SESSIONKEYBYTES];

/* Generate the server's key pair */
crypto_kx_keypair(server_pk, server_sk);

/* Prerequisite after this point: the client's public key must be known by the server
*/

/* Compute two shared keys using the client's public key and the server's secret key.
   server_rx will be used by the server to receive data from the client,
   server_tx will be used by the server to send data to the client. */
if (crypto_kx_server_session_keys(server_rx, server_tx,
                                  server_pk, server_sk, client_pk) != 0) {
    /* Suspicious client public key, bail out */
}
```

Purpose

Using the key exchange API, two parties can securely compute a set of shared keys using their peer's public key and their own secret key.

This API was introduced in libsodium 1.0.12.

Usage

```
int crypto_kx_keypair(unsigned char pk[crypto_kx_PUBLICKEYBYTES],
                     unsigned char sk[crypto_kx_SECRETKEYBYTES]);
```

The `crypto_kx_keypair()` function creates a new key pair. It puts the public key into `pk` and the secret key into `sk`.

```
int crypto_kx_seed_keypair(unsigned char pk[crypto_kx_PUBLICKEYBYTES],
                           unsigned char sk[crypto_kx_SECRETKEYBYTES],
                           const unsigned char seed[crypto_kx_SEEDBYTES]);
```

The `crypto_kx_seed_keypair()` function computes a deterministic key pair from the seed `seed` (`crypto_kx_SEEDBYTES` bytes).

```
int crypto_kx_client_session_keys(unsigned char rx[crypto_kx_SESSIONKEYBYTES],
                                  unsigned char tx[crypto_kx_SESSIONKEYBYTES],
                                  const unsigned char client_pk[crypto_kx_PUBLICKEYBYTES],
                                  const unsigned char client_sk[crypto_kx_SECRETKEYBYTES],
                                  const unsigned char server_pk[crypto_kx_PUBLICKEYBYTES]);
```

The `crypto_kx_client_session_keys()` function computes a pair of shared keys (`rx` and `tx`) using the client's public key `client_pk`, the client's secret key `client_sk` and the server's public key `server_pk`.

It returns `0` on success, or `-1` if the server's public key is not acceptable.

These keys can be used by any functions requiring secret keys up to `crypto_kx_SESSIONKEYBYTES` bytes, including `crypto_secretbox_*` and `crypto_aead_*`.

The shared secret key `rx` should be used by the client to receive data from the server, whereas `tx` should be used for data flowing in the opposite direction.

`rx` and `tx` are both `crypto_kx_SESSIONKEYBYTES` bytes long. If only one session key is required, either `rx` or `tx` can be set to `NULL`.

```
int crypto_kx_server_session_keys(unsigned char rx[crypto_kx_SESSIONKEYBYTES],
                                   unsigned char tx[crypto_kx_SESSIONKEYBYTES],
                                   const unsigned char server_pk[crypto_kx_PUBLICKEYBYTES],
                                   const unsigned char server_sk[crypto_kx_SECRETKEYBYTES],
                                   const unsigned char client_pk[crypto_kx_PUBLICKEYBYTES]);
```

The `crypto_kx_server_session_keys()` function computes a pair of shared keys (`rx` and `tx`) using the server's public key `server_pk`, the server's secret key `server_sk` and the client's public key `client_pk`.

It returns `0` on success, or `-1` if the client's public key is not acceptable.

The shared secret key `rx` should be used by the server to receive data from the client, whereas `tx` should be used for data flowing in the opposite direction.

`rx` and `tx` are both `crypto_kx_SESSIONKEYBYTES` bytes long. If only one session key is required, either `rx` or `tx` can be set to `NULL`.

Constants

- `crypto_kx_PUBLICKEYBYTES`
- `crypto_kx_SECRETKEYBYTES`
- `crypto_kx_SEEDBYTES`
- `crypto_kx_SESSIONKEYBYTES`
- `crypto_kx_PRIMITIVE`

Algorithm details

```
rx || tx = BLAKE2B-512(p.n || client_pk || server_pk)
```

Notes

For earlier versions of the library that didn't implement this API, the X25519 function is accessible directly using the `crypto_scalarmult_*` API.

Having different keys for each direction allows counters to be safely used as nonces without having to wait for an acknowledgement after every message.

Advanced

The functions detailed in this section are low-level, and implement specific algorithms.

They are only designed to be used as building blocks for custom constructions, or for interoperability with other libraries and applications.

As a result, using these functions directly may not be secure if not done correctly.

Low-level functions that are not required by high-level APIs are also not present in libsodium when compiled in minimal mode.

Unless you absolutely need these specific algorithms, use the high-level APIs whenever possible.

Bindings for 3rd party languages are encouraged to use the high-level APIs as well. The underlying functions they depend on is guaranteed to never change without a major bump of the library version.

The SHA-2 hash functions family

The SHA-256 and SHA-512 functions are provided for interoperability with other applications. If you are looking for a generic hash function and not specifically SHA-2, using `crypto_generichash()` (BLAKE2b) might be a better choice.

These functions are also not suitable for hashing passwords or deriving keys from passwords. Use one of the [password hashing](#) APIs instead.

These functions are not keyed and are thus deterministic. In addition, the untruncated versions are vulnerable to length extension attacks.

A message can be hashed in a single pass, but a streaming API is also available to process a message as a sequence of multiple chunks.

Single-part SHA-256 example

```
#define MESSAGE ((const unsigned char *) "test")
#define MESSAGE_LEN 4

unsigned char out[crypto_hash_sha256_BYTES];

crypto_hash_sha256(out, MESSAGE, MESSAGE_LEN);
```

Multi-part SHA-256 example

```
#define MESSAGE_PART1 \  
    ((const unsigned char *) "Arbitrary data to hash")  
#define MESSAGE_PART1_LEN 22  
  
#define MESSAGE_PART2 \  
    ((const unsigned char *) "is longer than expected")  
#define MESSAGE_PART2_LEN 23  
  
unsigned char out[crypto_hash_sha256_BYTES];  
crypto_hash_sha256_state state;  
  
crypto_hash_sha256_init(&state);  
  
crypto_hash_sha256_update(&state, MESSAGE_PART1, MESSAGE_PART1_LEN);  
crypto_hash_sha256_update(&state, MESSAGE_PART2, MESSAGE_PART2_LEN);  
  
crypto_hash_sha256_final(&state, out);
```

Usage

SHA-256

Single-part:

```
int crypto_hash_sha256(unsigned char *out, const unsigned char *in,  
                      unsigned long long inlen);
```

Multi-part:

```
int crypto_hash_sha256_init(crypto_hash_sha256_state *state);  
  
int crypto_hash_sha256_update(crypto_hash_sha256_state *state,  
                             const unsigned char *in,  
                             unsigned long long inlen);  
  
int crypto_hash_sha256_final(crypto_hash_sha256_state *state,  
                             unsigned char *out);
```

SHA-512

Single-part:

```
int crypto_hash_sha512(unsigned char *out, const unsigned char *in,  
                      unsigned long long inlen);
```

Multi-part:

```
int crypto_hash_sha512_init(crypto_hash_sha512_state *state);

int crypto_hash_sha512_update(crypto_hash_sha512_state *state,
                              const unsigned char *in,
                              unsigned long long inlen);

int crypto_hash_sha512_final(crypto_hash_sha512_state *state,
                             unsigned char *out);
```

Notes

The state must be initialized with `crypto_hash_sha*_init()` before updating or finalizing it.

After `crypto_hash_sha*_final()`, the state should not be used any more, unless it is reinitialized using `crypto_hash_sha*_init()`.

SHA-512-256 is also available via the higher-level interface `crypto_hash()`.

Constants

- `crypto_hash_sha256_BYTES`
- `crypto_hash_sha512_BYTES`

Data types

- `crypto_hash_sha256_state`
- `crypto_hash_sha512_state`

HMAC-SHA-2

The keyed message authentication codes HMAC-SHA-256, HMAC-SHA-512 and HMAC-SHA512-256 (truncated HMAC-SHA-512) are provided.

The `crypto_auth` API provides a simplified interface for message authentication.

If required, a streaming API is available to process a message as a sequence of multiple chunks.

Single-part example

```
#define MESSAGE ((const unsigned char *) "Arbitrary data to hash")
#define MESSAGE_LEN 22

unsigned char hash[crypto_auth_hmacsha512_BYTES];
unsigned char key[crypto_auth_hmacsha512_KEYBYTES];

crypto_auth_hmacsha512_keygen(key);
crypto_auth_hmacsha512(hash, MESSAGE, MESSAGE_LEN, key);
```

Multi-part example

```
#define MESSAGE_PART1 \  
    ((const unsigned char *) "Arbitrary data to hash")  
#define MESSAGE_PART1_LEN 22  
  
#define MESSAGE_PART2 \  
    ((const unsigned char *) "is longer than expected")  
#define MESSAGE_PART2_LEN 23  
  
unsigned char hash[crypto_auth_hmacsha512_BYTES];  
unsigned char key[crypto_auth_hmacsha512_KEYBYTES];  
crypto_auth_hmacsha512_state state;  
  
crypto_auth_hmacsha512_keygen(key);  
  
crypto_auth_hmacsha512_init(&state, key, sizeof key);  
  
crypto_auth_hmacsha512_update(&state, MESSAGE_PART1, MESSAGE_PART1_LEN);  
crypto_auth_hmacsha512_update(&state, MESSAGE_PART2, MESSAGE_PART2_LEN);  
  
crypto_auth_hmacsha512_final(&state, hash);
```

Usage

HMAC-SHA-256

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

The `crypto_auth_hmacsha256()` function authenticates a message `in` whose length is `inlen` using the secret key `k` whose length is `crypto_auth_hmacsha256_KEYBYTES`, and puts the authenticator into `out` (`crypto_auth_hmacsha256_BYTES` bytes).

```
int crypto_auth_hmacsha256_verify(const unsigned char *h,  
                                  const unsigned char *in,  
                                  unsigned long long inlen,  
                                  const unsigned char *k);
```

The `crypto_auth_hmacsha256_verify()` function verifies in constant time that `h` is a correct authenticator for the message `in` whose length is `inlen` under a secret key `k` (`crypto_auth_hmacsha256_KEYBYTES` bytes).

It returns `-1` if the verification fails, and `0` on success.

A multi-part (streaming) API can be used instead of `crypto_auth_hmacsha256()` :

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

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

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

This alternative API supports a key of arbitrary length `keylen` .

However, please note that in the HMAC construction, a key larger than the block size gets reduced to `h(key)` .

```
void crypto_auth_hmacsha256_keygen(unsigned char k[crypto_auth_hmacsha256_KEYBYTES]);
```

This helper function introduced in libsodium 1.0.12 creates a random key `k` .

It is equivalent to calling `randombytes_buf()` but improves code clarity and can prevent misuse by ensuring that the provided key length is always be correct.

HMAC-SHA-512

Similarly to the `crypto_auth_hmacsha256_*`() set of functions, the `crypto_auth_hmacsha512_*`() set of functions implements HMAC-SHA512:

```
int crypto_auth_hmacsha512(unsigned char *out,
                            const unsigned char *in,
                            unsigned long long inlen,
                            const unsigned char *k);
```

```
int crypto_auth_hmacsha512_verify(const unsigned char *h,
                                   const unsigned char *in,
                                   unsigned long long inlen,
                                   const unsigned char *k);
```

```
int crypto_auth_hmacsha512_init(crypto_auth_hmacsha512_state *state,
                                const unsigned char *key,
                                size_t keylen);
```

```
int crypto_auth_hmacsha512_update(crypto_auth_hmacsha512_state *state,
                                   const unsigned char *in,
                                   unsigned long long inlen);
```

```
int crypto_auth_hmacsha512_final(crypto_auth_hmacsha512_state *state,
                                  unsigned char *out);
```

```
void crypto_auth_hmacsha512_keygen(unsigned char k[crypto_auth_hmacsha512_KEYBYTES]);
```

HMAC-SHA-512-256

HMAC-SHA-512-256 is implemented as HMAC-SHA-512 with the output truncated to 256 bits. This is slightly faster than HMAC-SHA-256. Note that this construction is not the same as HMAC-SHA-512/256, which is HMAC using the SHA-512/256 function.

```
int crypto_auth_hmacsha512256(unsigned char *out,
                                const unsigned char *in,
                                unsigned long long inlen,
                                const unsigned char *k);
```

```
int crypto_auth_hmacsha512256_verify(const unsigned char *h,
                                       const unsigned char *in,
                                       unsigned long long inlen,
                                       const unsigned char *k);
```

```
int crypto_auth_hmacsha512256_init(crypto_auth_hmacsha512256_state *state,
                                    const unsigned char *key,
                                    size_t keylen);
```

```
int crypto_auth_hmacsha512256_update(crypto_auth_hmacsha512256_state *state,
                                       const unsigned char *in,
                                       unsigned long long inlen);
```

```
int crypto_auth_hmacsha512256_final(crypto_auth_hmacsha512256_state *state,
                                      unsigned char *out);
```



```
void crypto_auth_hmacsha512256_keygen(unsigned char k[crypto_auth_hmacsha512256_KEYBYTES]);
```

Constants

- `crypto_auth_hmacsha256_BYTES`
- `crypto_auth_hmacsha256_KEYBYTES`
- `crypto_auth_hmacsha512_BYTES`
- `crypto_auth_hmacsha512_KEYBYTES`
- `crypto_auth_hmacsha512256_BYTES`
- `crypto_auth_hmacsha512256_KEYBYTES`

Data types

- `crypto_auth_hmacsha256_state`
- `crypto_auth_hmacsha512_state`
- `crypto_auth_hmacsha512256_state`

Notes

- The state must be initialized with `crypto_auth_hmacsha*_init()` before updating or finalizing it. After `crypto_auth_hmacsha*_final()` returns, the state should not be used any more, unless it is reinitialized using `crypto_auth_hmacsha*_init()`.
- Arbitrary key lengths are supported using the multi-part interface.
- `crypto_auth_hmacsha256_*` can be used to create AWS HMAC-SHA256 request signatures.
- Only use these functions for interoperability with 3rd party services. For everything else, you should probably use `crypto_auth()` / `crypto_auth_verify()` or `crypto_generichash_*` instead.

The Script memory-hard function

As a conservative option, Sodium provides an implementation of the Script password hashing function.

Unless you have specific reasons to use script, you should instead consider the default function, [Argon2](#).

Example 1: key derivation

```
#define PASSWORD "Correct Horse Battery Staple"
#define KEY_LEN crypto_box_SEEDBYTES

unsigned char salt[crypto_pwhash_scryptsalsa208sha256_SALTBYTES];
unsigned char key[KEY_LEN];

randombytes_buf(salt, sizeof salt);

if (crypto_pwhash_scryptsalsa208sha256
    (key, sizeof key, PASSWORD, strlen(PASSWORD), salt,
     crypto_pwhash_scryptsalsa208sha256_OPSLIMIT_INTERACTIVE,
     crypto_pwhash_scryptsalsa208sha256_MEMLIMIT_INTERACTIVE) != 0) {
    /* out of memory */
}
```

Example 2: password storage

```
#define PASSWORD "Correct Horse Battery Staple"

char hashed_password[crypto_pwhash_scryptsalsa208sha256_STRBYTES];

if (crypto_pwhash_scryptsalsa208sha256_str
    (hashed_password, PASSWORD, strlen(PASSWORD),
     crypto_pwhash_scryptsalsa208sha256_OPSLIMIT_SENSITIVE,
     crypto_pwhash_scryptsalsa208sha256_MEMLIMIT_SENSITIVE) != 0) {
    /* out of memory */
}

if (crypto_pwhash_scryptsalsa208sha256_str_verify
    (hashed_password, PASSWORD, strlen(PASSWORD)) != 0) {
    /* wrong password */
}
```

Key derivation

```
int crypto_pwhash_scryptsalsa208sha256(unsigned char * const out,
                                         unsigned long long outlen,
                                         const char * const passwd,
                                         unsigned long long passwdlen,
                                         const unsigned char * const salt,
                                         unsigned long long opslimit,
                                         size_t memlimit);
```

The `crypto_pwhash_scryptsalsa208sha256()` function derives an `outlen` bytes long key from a password `passwd` whose length is `passwdlen` and a salt `salt` whose fixed length is `crypto_pwhash_scryptsalsa208sha256_SALTBYTES` bytes.

The computed key is stored into `out`. `out` (and hence `outlen`) should be at least `crypto_pwhash_scryptsalsa208sha256_BYTES_MIN` and at most `crypto_pwhash_scryptsalsa208sha256_BYTES_MAX` (~ 127 GB).

`passwd` (and hence `passwdlen`) should be at least `crypto_pwhash_scryptsalsa208sha256_PASSWD_MIN` and at most `crypto_pwhash_scryptsalsa208sha256_PASSWD_MAX`.

`opslimit` represents a maximum amount of computations to perform. Raising this number will make the function require more CPU cycles to compute a key. This number must be between `crypto_pwhash_scryptsalsa208sha256_OPSLIMIT_MIN` and `crypto_pwhash_scryptsalsa208sha256_OPSLIMIT_MAX`.

`memlimit` is the maximum amount of RAM that the function will use, in bytes. It is highly recommended to allow the function to use at least 16 megabytes. This number must be between `crypto_pwhash_scryptsalsa208sha256_MEMLIMIT_MIN` and `crypto_pwhash_scryptsalsa208sha256_MEMLIMIT_MAX`.

For interactive, online operations, `crypto_pwhash_scryptsalsa208sha256_OPSLIMIT_INTERACTIVE` and `crypto_pwhash_scryptsalsa208sha256_MEMLIMIT_INTERACTIVE` provide a safe base line for these two parameters. However, using higher values may improve security.

For highly sensitive data, `crypto_pwhash_scryptsalsa208sha256_OPSLIMIT_SENSITIVE` and `crypto_pwhash_scryptsalsa208sha256_MEMLIMIT_SENSITIVE` can be used as an alternative. But with these parameters, deriving a key takes about 2 seconds on a 2.8 Ghz Core i7 CPU and requires up to 1 gigabyte of dedicated RAM.

The `salt` should be unpredictable. `randombytes_buf()` is the easiest way to fill the `crypto_pwhash_scryptsalsa208sha256_SALTBYTES` bytes of the salt.

Keep in mind that in order to produce the same key from the same password, the same salt, and the same values for `opslimit` and `memlimit` have to be used. Therefore, these parameters have to be stored for each user.

The function returns `0` on success, and `-1` if the computation didn't complete, usually because the operating system refused to allocate the amount of requested memory.

Password storage

```
int crypto_pwhash_scryptsalsa208sha256_str(char out[crypto_pwhash_scryptsalsa208sha256_STRBYTES],  
                                             const char * const passwd,  
                                             unsigned long long passwdlen,  
                                             unsigned long long opslimit,  
                                             size_t memlimit);
```

The `crypto_pwhash_scryptsalsa208sha256_str()` function puts an ASCII encoded string into `out`, which includes:

- the result of a memory-hard, CPU-intensive hash function applied to the password `passwd` of length `passwdlen`
- the automatically generated salt used for the previous computation
- the other parameters required to verify the password: `opslimit` and `memlimit`.

`crypto_pwhash_scryptsalsa208sha256_OPSLIMIT_INTERACTIVE` and `crypto_pwhash_scryptsalsa208sha256_MEMLIMIT_INTERACTIVE` are safe baseline values to use for `opslimit` and `memlimit`.

The output string is zero-terminated, includes only ASCII characters and can be safely stored into SQL databases and other data stores. No extra information has to be stored in order to verify the password.

The function returns `0` on success and `-1` if it didn't complete successfully.

```
int crypto_pwhash_scryptsalsa208sha256_str_verify(const char str[crypto_pwhash_scryptsalsa208sha256_STRBYTES],  
                                                  const char * const passwd,  
                                                  unsigned long long passwdlen);
```

This function verifies that the password `str` is a valid password verification string (as generated by `crypto_pwhash_scryptsalsa208sha256_str()`) for `passwd` whose length is `passwdlen`.

`str` has to be zero-terminated.

It returns `0` if the verification succeeds, and `-1` on error.

Guidelines for choosing scrypt parameters

Start by determining how much memory the scrypt function can use. What will be the highest number of threads/processes evaluating the function simultaneously (ideally, no more than 1 per CPU core)? How much physical memory is guaranteed to be available?

`memlimit` should be a power of 2. Do not use anything less than 16 Mb, even for interactive use.

Then, a reasonable starting point for `opslimit` is `memlimit / 32`.

Measure how long the scrypt function needs in order to hash a password. If this it is way too long for your application, reduce `memlimit` and adjust `opslimit` using the above formula.

If the function is so fast that you can afford it to be more computationally intensive without any usability issues, increase `opslimit`.

For online use (e.g. login in on a website), a 1 second computation is likely to be the acceptable maximum.

For interactive use (e.g. a desktop application), a 5 second pause after having entered a password is acceptable if the password doesn't need to be entered more than once per session.

For non-interactive use and infrequent use (e.g. restoring an encrypted backup), an even slower computation can be an option.

But the best defense against brute-force password cracking remains using strong passwords. Libraries such as [passwdqc](#) can help enforce this.

Low-level scrypt API

The traditional, low-level scrypt API is also available:

```
int crypto_pwhash_scryptsalsa208sha256_ll(const uint8_t * passwd, size_t passwdlen,
                                           const uint8_t * salt, size_t saltlen,
                                           uint64_t N, uint32_t r, uint32_t p,
                                           uint8_t * buf, size_t buflen);
```

Please note that `r` is specified in kilobytes, and not in bytes as in the Sodium API.

Constants

- `crypto_pwhash_scryptsalsa208sha256_BYTES_MIN`
- `crypto_pwhash_scryptsalsa208sha256_BYTES_MAX`
- `crypto_pwhash_scryptsalsa208sha256_PASSWD_MIN`
- `crypto_pwhash_scryptsalsa208sha256_PASSWD_MAX`
- `crypto_pwhash_scryptsalsa208sha256_SALTBYTES`
- `crypto_pwhash_scryptsalsa208sha256_STRBYTES`
- `crypto_pwhash_scryptsalsa208sha256_STRPREFIX`
- `crypto_pwhash_scryptsalsa208sha256_OPSLIMIT_MIN`
- `crypto_pwhash_scryptsalsa208sha256_OPSLIMIT_MAX`
- `crypto_pwhash_scryptsalsa208sha256_MEMLIMIT_MIN`
- `crypto_pwhash_scryptsalsa208sha256_MEMLIMIT_MAX`
- `crypto_pwhash_scryptsalsa208sha256_OPSLIMIT_INTERACTIVE`
- `crypto_pwhash_scryptsalsa208sha256_MEMLIMIT_INTERACTIVE`
- `crypto_pwhash_scryptsalsa208sha256_OPSLIMIT_SENSITIVE`
- `crypto_pwhash_scryptsalsa208sha256_MEMLIMIT_SENSITIVE`

Notes

Do not forget to initialize the library with `sodium_init()` .

`crypto_pwhash_scryptsalsa208sha256_*` will still work without doing so, but possibly way slower.

Do not use constants (including `crypto_pwhash_scryptsalsa208sha256_OPSLIMIT_*` and `crypto_pwhash_scryptsalsa208sha256_MEMLIMIT_*`) in order to verify a password or produce a deterministic output. Save the parameters along with the hash instead.

For password verification, the recommended interface is

`crypto_pwhash_scryptsalsa208sha256_str()` and `crypto_pwhash_scryptsalsa208sha256_str_verify()` . The string produced by `crypto_pwhash_scryptsalsa208sha256_str()` already includes an algorithm identifier, as well as all the parameters (including the automatically generated salt) that have been used to hash the password. Subsequently, `crypto_pwhash_scryptsalsa208sha256_str_verify()` automatically decodes these parameters.

By doing so, passwords can be rehashed using different parameters if required later on.

Cleartext passwords should not stay in memory longer than needed.

It is highly recommended to use `sodium_mlock()` to lock memory regions storing cleartext passwords, and to call `sodium_munlock()` right after

`crypto_pwhash_scryptsalsa208sha256_str()` and
`crypto_pwhash_scryptsalsa208sha256_str_verify()` return.

`sodium_munlock()` overwrites the region with zeros before unlocking it, so it doesn't have to be done before calling this function.

By design, a password whose length is 65 bytes or more is reduced to `SHA-256(password)`. This can have security implications if the password is present in another password database using raw, unsalted SHA-256. Or when upgrading passwords previously hashed with unsalted SHA-256 to scrypt.

If this is a concern, passwords should be pre-hashed before being hashed using scrypt:

```
char prehashed_password[56];
crypto_generichash((unsigned char *) prehashed_password, 56,
    (const unsigned char *) password, strlen(password), NULL, 0);
crypto_pwhash_scryptsalsa208sha256_str(out, prehashed_password, 56, ...);
...
crypto_pwhash_scryptsalsa208sha256_str_verify(str, prehashed_password, 56);
```

Algorithm details

- [The scrypt Password-Based Key Derivation Function](#)

Point*scalar multiplication (X25519)

Sodium provides an API to multiply a point on the Curve25519 curve.

This can be used as a building block to construct key exchange mechanisms, or more generally to compute a public key from a secret key.

On current libsodium versions, you generally want to use the `crypto_kx` API for key exchange instead.

Usage

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

Given a user's secret key `n` (`crypto_scalarmult_SCALARBYTES` bytes), the `crypto_scalarmult_base()` function computes the user's public key and puts it into `q` (`crypto_scalarmult_BYTES` bytes).

`crypto_scalarmult_BYTES` and `crypto_scalarmult_SCALARBYTES` are provided for consistency, but it is safe to assume that `crypto_scalarmult_BYTES == crypto_scalarmult_SCALARBYTES` .

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

This function can be used to compute a shared secret `q` given a user's secret key and another user's public key.

`n` is `crypto_scalarmult_SCALARBYTES` bytes long, `p` and the output are `crypto_scalarmult_BYTES` bytes long.

`q` represents the X coordinate of a point on the curve. As a result, the number of possible keys is limited to the group size ($\approx 2^{252}$), which is smaller than the key space.

For this reason, and to mitigate subtle attacks due to the fact many (`p` , `n`) pairs produce the same result, using the output of the multiplication `q` directly as a shared key is not recommended.

A better way to compute a shared key is `h(q || pk1 || pk2)` , with `pk1` and `pk2` being the public keys.

By doing so, each party can prove what exact public key they intended to perform a key exchange with (for a given public key, 11 other public keys producing the same shared secret can be trivially computed).

This can be achieved with the following code snippet:

```
unsigned char client_publickey[crypto_box_PUBLICKEYBYTES];
unsigned char client_secretkey[crypto_box_SECRETKEYBYTES];
unsigned char server_publickey[crypto_box_PUBLICKEYBYTES];
unsigned char server_secretkey[crypto_box_SECRETKEYBYTES];
unsigned char scalarmult_q_by_client[crypto_scalarmult_BYTES];
unsigned char scalarmult_q_by_server[crypto_scalarmult_BYTES];
unsigned char sharedkey_by_client[crypto_generichash_BYTES];
unsigned char sharedkey_by_server[crypto_generichash_BYTES];
crypto_generichash_state h;

/* Create client's secret and public keys */
randombytes_buf(client_secretkey, sizeof client_secretkey);
crypto_scalarmult_base(client_publickey, client_secretkey);

/* Create server's secret and public keys */
randombytes_buf(server_secretkey, sizeof server_secretkey);
crypto_scalarmult_base(server_publickey, server_secretkey);

/* The client derives a shared key from its secret key and the server's public key */
/* shared key = h(q || client_publickey || server_publickey) */
if (crypto_scalarmult(scalarmult_q_by_client, client_secretkey, server_publickey) != 0
) {
    /* Error */
}
crypto_generichash_init(&h, NULL, 0U, sizeof sharedkey_by_client);
crypto_generichash_update(&h, scalarmult_q_by_client, sizeof scalarmult_q_by_client);
crypto_generichash_update(&h, client_publickey, sizeof client_publickey);
crypto_generichash_update(&h, server_publickey, sizeof server_publickey);
crypto_generichash_final(&h, sharedkey_by_client, sizeof sharedkey_by_client);
```

```

/* The server derives a shared key from its secret key and the client's public key */
/* shared key = h(q || client_publickey || server_publickey) */
if (crypto_scalarmult(scalarmult_q_by_server, server_secretkey, client_publickey) != 0
) {
    /* Error */
}
crypto_generichash_init(&h, NULL, 0U, sizeof sharedkey_by_server);
crypto_generichash_update(&h, scalarmult_q_by_server, sizeof scalarmult_q_by_server);
crypto_generichash_update(&h, client_publickey, sizeof client_publickey);
crypto_generichash_update(&h, server_publickey, sizeof server_publickey);
crypto_generichash_final(&h, sharedkey_by_server, sizeof sharedkey_by_server);

/* sharedkey_by_client and sharedkey_by_server are identical */

```

If the intent is to create 256-bit keys (or less) for encryption, the final hash can also be set to output 512 bits: the first half can be used as a key to encrypt in one direction (for example from the server to the client), and the other half can be used in the other direction.

When using counters as nonces, having distinct keys allows the client and the server to safely send multiple messages without having to wait from an acknowledgment after each message.

```

typedef struct kx_session_keypair {
    unsigned char rx[32];
    unsigned char tx[32];
} kx_session_keypair;

kx_session_keypair kp;

if (crypto_scalarmult(scalarmult_q_by_client, client_secretkey, server_publickey) != 0
) {
    /* Error */
}
crypto_generichash_init(&h, NULL, 0U, sizeof session_keypair_by_client);
crypto_generichash_update(&h, scalarmult_q_by_client, sizeof scalarmult_q_by_client);
crypto_generichash_update(&h, client_publickey, sizeof client_publickey);
crypto_generichash_update(&h, server_publickey, sizeof server_publickey);
crypto_generichash_final(&h, session_keypair_by_client, sizeof session_keypair_by_client);

```

`kp->tx` is a key that the server can use in order to encrypt data sent to the client, and `kp->rx` is a key that can be used in the opposite direction.

Constants

- `crypto_scalarmult_BYTES`

- `crypto_scalarmult_SCALARBYTES`

Algorithm details

- X25519 (ECDH over Curve25519) - [RFC 7748](#)

Secret-key single-message authentication using Poly1305

One-time authentication in Sodium uses Poly1305, a Wegman-Carter authenticator designed by D. J. Bernstein.

Poly1305 takes a 32-byte, one-time key and a message and produces a 16-byte tag that authenticates the message such that an attacker has a negligible chance of producing a valid tag for a inauthentic message.

Poly1305 keys have to be:

- secret. An attacker can compute a valid authentication tag for any message, for any given key. The security of Poly1305 relies on the fact that attackers don't know the key being used to compute the tag. This implies that they have to be:
- unpredictable. Do not use timestamps or counters.
- unique. Never reuse a key. A new key is required for every single message. The key can be recovered if two messages are authenticated with the same key.

The standard way to use Poly1305's is to derive a dedicated subkey from a (key, nonce) tuple, for example by taking the first bytes generated by a stream cipher.

Due to its output size, Poly1305 is recommended for online protocols, exchanging many small messages, rather than for authenticating very large files.

Finally, Poly1305 is not a replacement for a hash function.

Single-part example

```
#define MESSAGE ((const unsigned char *) "Data to authenticate")
#define MESSAGE_LEN 20

unsigned char out[crypto_onetimeauth_BYTES];
unsigned char key[crypto_onetimeauth_KEYBYTES];

crypto_onetimeauth_keygen(key);
crypto_onetimeauth(out, MESSAGE, MESSAGE_LEN, key);

if (crypto_onetimeauth_verify(out, MESSAGE, MESSAGE_LEN, key) != 0) {
    /* message forged! */
}
```

Multi-part example

```
#define MESSAGE1 ((const unsigned char *) "Multi-part")
#define MESSAGE1_LEN 10
#define MESSAGE2 ((const unsigned char *) "data")
#define MESSAGE2_LEN 4

unsigned char out[crypto_onetimeauth_BYTES];
unsigned char key[crypto_onetimeauth_KEYBYTES];
crypto_onetimeauth_state state;

crypto_onetimeauth_keygen(key);

crypto_onetimeauth_init(&state, key);
crypto_onetimeauth_update(&state, MESSAGE1, MESSAGE1_LEN);
crypto_onetimeauth_update(&state, MESSAGE2, MESSAGE2_LEN);
crypto_onetimeauth_final(&state, out);
```

Usage

Single-part interface

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

The `crypto_onetimeauth()` function authenticates a message `in` whose length is `inlen` using a secret key `k` (`crypto_onetimeauth_KEYBYTES` bytes) and puts the authenticator into `out` (`crypto_onetimeauth_BYTES` bytes).

```
int crypto_onetimeauth_verify(const unsigned char *h, const unsigned char *in,
                             unsigned long long inlen, const unsigned char *k);
```

The `crypto_onetimeauth_verify()` function verifies, in constant time, that `h` is a correct authenticator for the message `in` whose length is `inlen` bytes, using the secret key `k`.

It returns `-1` if the verification fails, or `0` on success.

Multi-part (streaming) interface

```
int crypto_onetimeauth_init(crypto_onetimeauth_state *state,
                           const unsigned char *key);
```

```
int crypto_onetimeauth_update(crypto_onetimeauth_state *state,
                             const unsigned char *in,
                             unsigned long long inlen);
```

```
int crypto_onetimeauth_final(crypto_onetimeauth_state *state,
                             unsigned char *out);
```

The `crypto_onetimeauth_init()` function initializes a structure pointed by `state` using a key `key` .

A 16 bytes alignment is required for the address of `state` . The size of this value can be obtained using `sizeof(crypto_onetimeauth_state)` , OR `crypto_onetimeauth_statebytes()` .

`crypto_onetimeauth_update()` can then be called more than one in order to compute the authenticator from sequential chunks of the message.

Finally, `crypto_onetimeauth_final()` puts the authenticator into `out` .

The state must be initialized with `crypto_onetimeauth_init()` before updating or finalizing it.

After `crypto_onetimeauth_final()` returns, the state should not be used any more, unless it is reinitialized using `crypto_onetimeauth_init()` .

```
void crypto_onetimeauth_keygen(unsigned char k[crypto_onetimeauth_KEYBYTES]);
```

The `crypto_onetimeauth_keygen()` function fills `k` with a random key. This convenience function was introduced in libsodium 1.0.12.

Constants

- `crypto_onetimeauth_BYTES`
- `crypto_onetimeauth_KEYBYTES`

Data types

- `crypto_onetimeauth_state`

Algorithm details

- Poly1305

Stream ciphers

Sodium includes implementations of the Salsa20, XSalsa20, ChaCha20 and XChaCha20 stream ciphers.

These functions are stream ciphers. They do not provide authenticated encryption.

They can be used to generate pseudo-random data from a key, or as building blocks for implementing custom constructions, but they are not alternatives to `crypto_secretbox_*` .

ChaCha20

ChaCha20 is a stream cipher developed by Daniel J. Bernstein. Its original design expands a 256-bit key into 2^{64} randomly accessible streams, each containing 2^{64} randomly accessible 64-byte (512 bits) blocks. It is a variant of Salsa20 with better diffusion.

ChaCha20 doesn't require any lookup tables and avoids the possibility of timing attacks.

Internally, ChaCha20 works like a block cipher used in counter mode. It includes an internal block counter to avoid incrementing the nonce after each block.

Two variants of the ChaCha20 cipher are implemented in libsodium:

- The original ChaCha20 cipher with a 64-bit nonce and a 64-bit counter, allowing a practically unlimited amount of data to be encrypted with the same (key, nonce) pair.
- The IETF variant increases the nonce size to 96 bits, but reduces the counter size down to 32 bits, allowing only up to 256 GB of data to be safely encrypted with a given (key, nonce) pair.

These primitives should only be used to implement protocols that specifically require them.

For all other applications, it is recommended to use the high-level `crypto_stream` API (XSalsa20) or the ChaCha20-based construction with an extended nonce, XChaCha20 (`crypto_stream_xchacha20`).

Usage (original construction)

```
int crypto_stream_chacha20(unsigned char *c, unsigned long long clen,
                           const unsigned char *n, const unsigned char *k);
```

The `crypto_stream_chacha20()` function stores `clen` pseudo random bytes into `c` using a nonce `n` (`crypto_stream_chacha20_NONCEBYTES` bytes) and a secret key `k` (`crypto_stream_chacha20_KEYBYTES` bytes).

```
int crypto_stream_chacha20_xor(unsigned char *c, const unsigned char *m,
                                unsigned long long mlen, const unsigned char *n,
                                const unsigned char *k);
```

The `crypto_stream_chacha20_xor()` function encrypts a message `m` of length `mlen` using a nonce `n` (`crypto_stream_chacha20_NONCEBYTES` bytes) and a secret key `k` (`crypto_stream_chacha20_KEYBYTES` bytes).

The ciphertext is put into `c`. The ciphertext is the message combined with the output of the stream cipher using the XOR operation, and doesn't include any authentication tag.

`m` and `c` can point to the same address (in-place encryption/decryption). If they don't, the regions should not overlap.

```
int crypto_stream_chacha20_xor_ic(unsigned char *c, const unsigned char *m,
                                   unsigned long long mlen,
                                   const unsigned char *n, uint64_t ic,
                                   const unsigned char *k);
```

The `crypto_stream_chacha20_xor_ic()` function is similar to `crypto_stream_chacha20_xor()` but adds the ability to set the initial value of the block counter to a non-zero value, `ic`.

This permits direct access to any block without having to compute the previous ones.

`m` and `c` can point to the same address (in-place encryption/decryption). If they don't, the regions should not overlap.

Usage (IETF variant, message length limited to 256 GB)

```
int crypto_stream_chacha20_ietf(unsigned char *c, unsigned long long clen,
                                  const unsigned char *n, const unsigned char *k);
```

The `crypto_stream_chacha20_ietf()` function stores `clen` pseudo random bytes into `c` using a nonce `n` (`crypto_stream_chacha20_ietf_NONCEBYTES` bytes) and a secret key `k` (`crypto_stream_chacha20_ietf_KEYBYTES` bytes).

```
int crypto_stream_chacha20_ietf_xor(unsigned char *c, const unsigned char *m,
                                      unsigned long long mlen, const unsigned char *n,
                                      const unsigned char *k);
```

The `crypto_stream_chacha20_ietf_xor()` function encrypts a message `m` of length `mlen` using a nonce `n` (`crypto_stream_chacha20_ietf_NONCEBYTES` bytes) and a secret key `k` (`crypto_stream_chacha20_ietf_KEYBYTES` bytes).

The ciphertext is put into `c`. The ciphertext is the message combined with the output of the stream cipher using the XOR operation, and doesn't include any authentication tag.

`m` and `c` can point to the same address (in-place encryption/decryption). If they don't, the regions should not overlap.

```
int crypto_stream_chacha20_ietf_xor_ic(unsigned char *c, const unsigned char *m,
                                         unsigned long long mlen,
                                         const unsigned char *n, uint32_t ic,
                                         const unsigned char *k);
```

The `crypto_stream_chacha20_ietf_xor_ic()` function is similar to `crypto_stream_chacha20_ietf_xor()` but adds the ability to set the initial value of the block counter to a non-zero value, `ic`.

This permits direct access to any block without having to compute the previous ones.

`m` and `c` can point to the same address (in-place encryption/decryption). If they don't, the regions should not overlap.

Constants

- `crypto_stream_chacha20_KEYBYTES`
- `crypto_stream_chacha20_NONCEBYTES`
- `crypto_stream_chacha20_ietf_KEYBYTES`
- `crypto_stream_chacha20_ietf_NONCEBYTES`

Notes

The nonce is short. In order to prevent nonce reuse, if a key is being reused, it is recommended to increment the previous nonce instead of generating a random nonce every time a new stream is required.

With the IETF variant, up to 256 GB can be produced from the a (`key` , `nonce`) pair. The original design doesn't have this limitation.

XChaCha20

XChaCha20 is a variant of ChaCha20 with an extended nonce, allowing random nonces to be safe.

XChaCha20 doesn't require any lookup tables and avoids the possibility of timing attacks.

Internally, XChaCha20 works like a block cipher used in counter mode. It uses the HChaCha20 hash function to derive a subkey and a subnonce from the original key and extended nonce, and a dedicated 64-bit block counter to avoid incrementing the nonce after each block.

XChaCha20 is generally recommended over plain ChaCha20 due to its extended nonce size, and its comparable performance. However, XChaCha20 is currently not widely implemented outside the libsodium library, due to the absence of formal specification.

Usage

```
int crypto_stream_xchacha20(unsigned char *c, unsigned long long clen,
                             const unsigned char *n, const unsigned char *k);
```

The `crypto_stream_xchacha20()` function stores `clen` pseudo random bytes into `c` using a nonce `n` (`crypto_stream_xchacha20_NONCEBYTES` bytes) and a secret key `k` (`crypto_stream_xchacha20_KEYBYTES` bytes).

```
int crypto_stream_xchacha20_xor(unsigned char *c, const unsigned char *m,
                                 unsigned long long mlen, const unsigned char *n,
                                 const unsigned char *k);
```

The `crypto_stream_xchacha20_xor()` function encrypts a message `m` of length `mlen` using a nonce `n` (`crypto_stream_xchacha20_NONCEBYTES` bytes) and a secret key `k` (`crypto_stream_xchacha20_KEYBYTES` bytes).

The ciphertext is put into `c`. The ciphertext is the message combined with the output of the stream cipher using the XOR operation, and doesn't include any authentication tag.

`m` and `c` can point to the same address (in-place encryption/decryption). If they don't, the regions should not overlap.

```
int crypto_stream_xchacha20_xor_ic(unsigned char *c, const unsigned char *m,
                                   unsigned long long mlen,
                                   const unsigned char *n, uint64_t ic,
                                   const unsigned char *k);
```

The `crypto_stream_xchacha20_xor_ic()` function is similar to `crypto_stream_xchacha20_xor()` but adds the ability to set the initial value of the block counter to a non-zero value, `ic`.

This permits direct access to any block without having to compute the previous ones.

`m` and `c` can point to the same address (in-place encryption/decryption). If they don't, the regions should not overlap.

```
void crypto_stream_xchacha20_keygen(unsigned char k[crypto_stream_xchacha20_KEYBYTES])
;
```

This helper function introduced in libsodium 1.0.12 creates a random key `k`.

It is equivalent to calling `randombytes_buf()` but improves code clarity and can prevent misuse by ensuring that the provided key length is always be correct.

Constants

- `crypto_stream_xchacha20_KEYBYTES`
- `crypto_stream_xchacha20_NONCEBYTES`

Notes

Unlike plain ChaCha20, the nonce is 192 bits long, so that generating a random nonce for every message is safe. If the output of the PRNG is indistinguishable from random data, the probability for a collision to happen is negligible.

XChaCha20 was implemented in libsodium 1.0.12.

Salsa20

Salsa20 is a stream cipher developed by Daniel J. Bernstein that expands a 256-bit key into 2^{64} randomly accessible streams, each containing 2^{64} randomly accessible 64-byte (512 bits) blocks.

Salsa20 doesn't require any lookup tables and avoids the possibility of timing attacks.

Internally, Salsa20 works like a block cipher used in counter mode. It uses a dedicated 64-bit block counter to avoid incrementing the nonce after each block.

The extended-nonce construction XSalsa20 is generally recommended over raw Salsa20, as it makes it easier to safely generate nonces.

Usage

```
int crypto_stream_salsa20(unsigned char *c, unsigned long long clen,
                          const unsigned char *n, const unsigned char *k);
```

The `crypto_stream_salsa20()` function stores `clen` pseudo random bytes into `c` using a nonce `n` (`crypto_stream_salsa20_NONCEBYTES` bytes) and a secret key `k` (`crypto_stream_salsa20_KEYBYTES` bytes).

```
int crypto_stream_salsa20_xor(unsigned char *c, const unsigned char *m,
                              unsigned long long mlen, const unsigned char *n,
                              const unsigned char *k);
```

The `crypto_stream_salsa20_xor()` function encrypts a message `m` of length `mlen` using a nonce `n` (`crypto_stream_salsa20_NONCEBYTES` bytes) and a secret key `k` (`crypto_stream_salsa20_KEYBYTES` bytes).

The ciphertext is put into `c`. The ciphertext is the message combined with the output of the stream cipher using the XOR operation, and doesn't include any authentication tag.

`m` and `c` can point to the same address (in-place encryption/decryption). If they don't, the regions should not overlap.

```
int crypto_stream_salsa20_xor_ic(unsigned char *c, const unsigned char *m,
                                unsigned long long mlen,
                                const unsigned char *n, uint64_t ic,
                                const unsigned char *k);
```

The `crypto_stream_salsa20_xor_ic()` function is similar to `crypto_stream_salsa20_xor()` but adds the ability to set the initial value of the block counter to a non-zero value, `ic`.

This permits direct access to any block without having to compute the previous ones.

`m` and `c` can point to the same address (in-place encryption/decryption). If they don't, the regions should not overlap.

```
void crypto_stream_salsa20_keygen(unsigned char k[crypto_stream_salsa20_KEYBYTES]);
```

This helper function introduced in libsodium 1.0.12 creates a random key `k`.

It is equivalent to calling `randombytes_buf()` but improves code clarity and can prevent misuse by ensuring that the provided key length is always be correct.

Constants

- `crypto_stream_salsa20_KEYBYTES`
- `crypto_stream_salsa20_NONCEBYTES`

Notes

The nonce is 64 bits long. In order to prevent nonce reuse, if a key is being reused, it is recommended to increment the previous nonce instead of generating a random nonce every time a new stream is required.

Alternatively, XSalsa20, a variant of Salsa20 with a longer nonce, can be used.

The functions described above perform 20 rounds of Salsa20.

Faster, reduced-rounds versions are also available:

Salsa20 reduced to 12 rounds

```
int crypto_stream_salsa2012(unsigned char *c, unsigned long long clen,
                             const unsigned char *n, const unsigned char *k);

int crypto_stream_salsa2012_xor(unsigned char *c, const unsigned char *m,
                                 unsigned long long mlen, const unsigned char *n,
                                 const unsigned char *k);

void crypto_stream_salsa2012_keygen(unsigned char k[crypto_stream_salsa2012_KEYBYTES])
;
```

Salsa20 reduced to 8 rounds

```
int crypto_stream_salsa208(unsigned char *c, unsigned long long clen,
                             const unsigned char *n, const unsigned char *k);

int crypto_stream_salsa208_xor(unsigned char *c, const unsigned char *m,
                                 unsigned long long mlen, const unsigned char *n,
                                 const unsigned char *k);

void crypto_stream_salsa208_keygen(unsigned char k[crypto_stream_salsa208_KEYBYTES]);
```

Although the best known attack against Salsa20-8 is not practical, the full-round version provides a highest security margin while still being fast enough for most purposes.

XSalsa20

XSalsa20 is a stream cipher based upon Salsa20 but with a much longer nonce: 192 bits instead of 64 bits.

XSalsa20 uses a 256-bit key as well as the first 128 bits of the nonce in order to compute a subkey. This subkey, as well as the remaining 64 bits of the nonce, are the parameters of the Salsa20 function used to actually generate the stream.

Like Salsa20, XSalsa20 is immune to timing attacks and provides its own 64-bit block counter to avoid incrementing the nonce after each block.

But with XSalsa20's longer nonce, it is safe to generate nonces using `randombytes_buf()` for every message encrypted with the same key without having to worry about a collision.

Sodium exposes XSalsa20 with 20 rounds as the `crypto_stream` operation.

Usage

```
int crypto_stream(unsigned char *c, unsigned long long clen,
                  const unsigned char *n, const unsigned char *k);
```

The `crypto_stream()` function stores `clen` pseudo random bytes into `c` using a nonce `n` (`crypto_stream_NONCEBYTES` bytes) and a secret key `k` (`crypto_stream_KEYBYTES` bytes).

```
int crypto_stream_xor(unsigned char *c, const unsigned char *m,
                     unsigned long long mlen, const unsigned char *n,
                     const unsigned char *k);
```

The `crypto_stream_xor()` function encrypts a message `m` of length `mlen` using a nonce `n` (`crypto_stream_NONCEBYTES` bytes) and a secret key `k` (`crypto_stream_KEYBYTES` bytes).

The ciphertext is put into `c`. The ciphertext is the message combined with the output of the stream cipher using the XOR operation, and doesn't include any authentication tag.

`m` and `c` can point to the same address (in-place encryption/decryption). If they don't, the regions should not overlap.

```
void crypto_stream_keygen(unsigned char k[crypto_stream_KEYBYTES]);
```

This helper function introduced in libsodium 1.0.12 creates a random key `k` .

It is equivalent to calling `randombytes_buf()` but improves code clarity and can prevent misuse by ensuring that the provided key length is always be correct.

Constants

- `crypto_stream_KEYBYTES`
- `crypto_stream_NONCEBYTES`
- `crypto_stream_PRIMITIVE`

Ed25519 to X25519 keys conversion

Ed25519 keys can be converted to X25519 keys, so that the same key pair can be used both for authenticated encryption (`crypto_box`) and for signatures (`crypto_sign`).

Example

```
unsigned char ed25519_pk[crypto_sign_ed25519_PUBLICKEYBYTES];
unsigned char ed25519_skpk[crypto_sign_ed25519_SECRETKEYBYTES];
unsigned char x25519_pk[crypto_scalarmult_curve25519_BYTES];
unsigned char x25519_sk[crypto_scalarmult_curve25519_BYTES];

crypto_sign_ed25519_keypair(ed25519_pk, ed25519_skpk);

crypto_sign_ed25519_pk_to_curve25519(x25519_pk, ed25519_pk);
crypto_sign_ed25519_sk_to_curve25519(x25519_sk, ed25519_skpk);
```

Usage

```
int crypto_sign_ed25519_pk_to_curve25519(unsigned char *x25519_pk,
                                           const unsigned char *ed25519_pk);
```

The `crypto_sign_ed25519_pk_to_curve25519()` function converts an Ed25519 public key `ed25519_pk` to an X25519 public key and stores it into `x25519_pk` .

```
int crypto_sign_ed25519_sk_to_curve25519(unsigned char *x25519_sk,
                                           const unsigned char *ed25519_sk);
```

The `crypto_sign_ed25519_sk_to_curve25519()` function converts an Ed25519 secret key `ed25519_sk` to an X25519 secret key and stores it into `x25519_sk` .

In order to save some CPU cycles, the `crypto_sign_open()` and `crypto_sign_verify_detached()` functions expect the secret key to be followed by the public key, as generated by `crypto_sign_keypair()` and `crypto_sign_seed_keypair()` .

However, the `crypto_sign_ed25519_sk_to_curve25519()` function doesn't have this requirement, and it is perfectly fine to provide only the Ed25519 secret key to this function.

Notes

If you can afford it, using distinct keys for signing and for encryption is still highly recommended.

Finite field arithmetic

A set of low-level APIs to perform computations over the edwards25519 curve, only useful to implement custom constructions.

Points are represented as their Y coordinate.

Example

Perform a secure two-party computation of $f(x) = p(x)^k$. x is the input sent to the second party by the first party after blinding it using a random invertible scalar r , and k is a secret key only known by the second party. $p(x)$ is a hash-to-curve function.

```

// ----- First party ----- Send blinded p(x)
unsigned char x[crypto_core_ed25519_UNIFORMBYTES];
randombytes_buf(x, sizeof x);

// Compute px = p(x), an EC point representative for x
unsigned char px[crypto_core_ed25519_BYTES];
crypto_core_ed25519_from_uniform(px, x);

// Compute a = p(x) * g^r
unsigned char r[crypto_core_ed25519_SCALARBYTES];
unsigned char gr[crypto_core_ed25519_BYTES];
unsigned char a[crypto_core_ed25519_BYTES];
crypto_core_ed25519_scalar_random(r);
crypto_scalarmult_ed25519_base_noclamp(gr, r);
crypto_core_ed25519_add(a, px, gr);

// ----- Second party ----- Send g^k and a^k
unsigned char k[crypto_core_ed25519_SCALARBYTES];
randombytes_buf(k, sizeof k);

// Compute v = g^k
unsigned char v[crypto_core_ed25519_BYTES];
crypto_scalarmult_ed25519_base(v, k);

// Compute b = a^k
unsigned char b[crypto_core_ed25519_BYTES];
if (crypto_scalarmult_ed25519(b, k, a) != 0) {
    return -1;
}

// ----- First party ----- Unblind f(x)
// Compute vir = v^(-r)
unsigned char ir[crypto_core_ed25519_SCALARBYTES];
unsigned char vir[crypto_core_ed25519_BYTES];
crypto_core_ed25519_scalar_negate(ir, r);
crypto_scalarmult_ed25519_noclamp(vir, ir, v);

// Compute f(x) = b * v^(-r) = (p(x) * g^r)^k * (g^k)^(-r)
//                = (p(x) * g)^k * g^(-k) = p(x)^k
unsigned char fx[crypto_core_ed25519_BYTES];
crypto_core_ed25519_add(fx, b, vir);

```

Point validation

```

int crypto_core_ed25519_is_valid_point(const unsigned char *p);

```

The `crypto_core_ed25519_is_valid_point()` function checks that `p` represents a point on the edwards25519 curve, in canonical form, on the main subgroup, and that the point doesn't have a small order.

It returns `1` on success, and `0` if the checks didn't pass.

Random group element

```
void crypto_core_ed25519_random(unsigned char *p);
```

Fills `p` with the representation of a random group element.

Elligator 2 map

```
int crypto_core_ed25519_from_uniform(unsigned char *p, const unsigned char *r);
```

The `crypto_core_ed25519_from_uniform()` function maps a 32 bytes vector `r` to a point, and stores its compressed representation into `p`.

The point is guaranteed to be on the main subgroup.

This function directly exposes the inverse Elligator 2 map, uses the high bit to set the sign of the X coordinate, and the resulting point is multiplied by the cofactor.

Hash-to-group

Scalar multiplication

```
int crypto_scalarmult_ed25519(unsigned char *q, const unsigned char *n,
                               const unsigned char *p);
```

The `crypto_scalarmult_ed25519()` function multiplies a point `p` by a scalar `n` and puts the Y coordinate of the resulting point into `q`.

`q` should not be used as a shared key prior to hashing.

The function returns `0` on success, or `-1` if `n` is `0` or if `p` is not on the curve, not on the main subgroup, is a point of small order, or is not provided in canonical form.

Note that `n` is "clamped" (the 3 low bits are cleared to make it a multiple of the cofactor, bit 254 is set and bit 255 is cleared to respect the original design).

```
int crypto_scalarmult_ed25519_base(unsigned char *q, const unsigned char *n);
```

The `crypto_scalarmult_ed25519_base()` function multiplies the base point $(x, 4/5)$ by a scalar `n` (clamped) and puts the Y coordinate of the resulting point into `q`.

The function returns `-1` if `n` is `0`, and `0` otherwise.

Scalar multiplication without clamping

In order to prevent attacks using small subgroups, the `scalarmult` functions above clear lower bits of the scalar. This may be undesirable to build protocols that requires `n` to be invertible.

The `noclamp` variants of these functions do not clear these bits, and do not set the high bit either. These variants expect a scalar in the `[0..L]` range.

```
int crypto_scalarmult_ed25519_noclamp(unsigned char *q, const unsigned char *n,  
                                     const unsigned char *p);
```

The function verifies that `p` is on the prime-order subgroup before performing the multiplication, and return `-1` if this is not the case or `n` is `0`. It returns `0` on success.

```
int crypto_scalarmult_ed25519_base_noclamp(unsigned char *q, const unsigned char *n);
```

The function returns `0` on success, or `-1` if `n` is `0`.

Point addition/substraction

```
int crypto_core_ed25519_add(unsigned char *r,  
                           const unsigned char *p, const unsigned char *q);
```

The `crypto_core_ed25519_add()` function adds the point `p` to the point `q` and stores the resulting point into `r`.

The function returns `0` on success, or `-1` if `p` and/or `q` are not valid points.


```
int crypto_core_ed25519_sub(unsigned char *r,
                             const unsigned char *p, const unsigned char *q);
```

The `crypto_core_ed25519_sub()` function subtracts the point `p` to the point `q` and stores the resulting point into `r`.

The function returns `0` on success, or `-1` if `p` and/or `q` are not valid points.

Scalar arithmetic over L

Scalars should ideally be randomly chosen in the `[0..L[` interval, `L` being the order of the main subgroup ($2^{252} + 27742317777372353535851937790883648493$).

This can be achieved with the following function, introduced in libsodium 1.0.17:

```
void crypto_core_ed25519_scalar_random(unsigned char *r);
```

`crypto_core_ed25519_scalar_random()` fills `r` with a `crypto_core_ed25519_SCALARBYTES` bytes representation of the scalar in the `]0..L[` interval.

A scalar in the `[0..L[` interval can also be obtained by reducing a possibly larger value:

```
void crypto_core_ed25519_scalar_reduce(unsigned char *r, const unsigned char *s);
```

The `crypto_core_ed25519_scalar_reduce()` function reduces `s` to `s mod L` and puts the `crypto_core_ed25519_SCALARBYTES` integer into `r`.

Note that `s` is much larger than `r` (64 bytes vs 32 bytes). Bits of `s` can be left to `0`, but the interval `s` is sampled from should be at least 317 bits to ensure almost uniformity of `r` over `L`.

```
int crypto_core_ed25519_scalar_invert(unsigned char *recip, const unsigned char *s);
```

The `crypto_core_ed25519_scalar_invert()` function computes the multiplicative inverse of `s` over `L`, and puts it into `recip`.

```
void crypto_core_ed25519_scalar_negate(unsigned char *neg, const unsigned char *s);
```

The `crypto_core_ed25519_scalar_negate()` function returns `neg` so that `s + neg = 0 (mod L)`.

```
void crypto_core_ed25519_scalar_complement(unsigned char *comp, const unsigned char *s)
;
```



The `crypto_core_ed25519_scalar_complement()` function returns `comp` so that $s + \text{comp} = 1 \pmod{L}$.

```
void crypto_core_ed25519_scalar_add(unsigned char *z,
                                     const unsigned char *x, const unsigned char *y);
```

The `crypto_core_ed25519_scalar_add()` function stores $x + y \pmod{L}$ into `z`.

```
void crypto_core_ed25519_scalar_sub(unsigned char *z,
                                     const unsigned char *x, const unsigned char *y);
```

The `crypto_core_ed25519_scalar_sub()` function stores $x - y \pmod{L}$ into `z`.

```
void crypto_core_ed25519_scalar_mul(unsigned char *z,
                                     const unsigned char *x, const unsigned char *y);
```

The `crypto_core_ed25519_scalar_mul()` function stores $x * y \pmod{L}$ into `z`.

Constants

- `crypto_scalarmult_ed25519_BYTES`
- `crypto_scalarmult_ed25519_SCALARBYTES`
- `crypto_core_ed25519_BYTES`
- `crypto_core_ed25519_UNIFORMBYTES`
- `crypto_core_ed25519_SCALARBYTES`
- `crypto_core_ed25519_NONREDUCEDSCALARBYTES`

Note

These functions were introduced in libsodium 1.0.16, 1.0.17 and 1.0.18.

For a complete example using these functions, see the [SPAKE2+EE implementation](#) for libsodium.

`crypto_core_ed25519_from_uniform()` exposes the Elligator 2 inverse map, using the high bit for the sign of the X coordinate.

Ristretto255

Ristretto is a new unified point compression format for curves over large-characteristic fields, which divides the curve's cofactor by 4 or 8 at very little cost of performance, efficiently implementing a prime-order group.

libsodium 1.0.18+ implements ristretto255: ristretto on top of the Curve25519 curve.

Compared to Curve25519 points encoded as their coordinates, ristretto makes it easier to safely implement protocols originally designed for prime-order groups.

Example

Perform a secure two-party computation of $f(x) = p(x)^k$. x is the input sent to the second party by the first party after blinding it using a random invertible scalar r , and k is a secret key only known by the second party. $p(x)$ is a hash-to-group function.

```

// ----- First party ----- Send blinded p(x)
unsigned char x[crypto_core_ristretto255_HASHBYTES];
randombytes_buf(x, sizeof x);

// Compute px = p(x), a group element derived from x
unsigned char px[crypto_core_ristretto255_BYTES];
crypto_core_ristretto255_from_hash(px, x);

// Compute a = p(x) * g^r
unsigned char r[crypto_core_ristretto255_SCALARBYTES];
unsigned char gr[crypto_core_ristretto255_BYTES];
unsigned char a[crypto_core_ristretto255_BYTES];
crypto_core_ristretto255_scalar_random(r);
crypto_scalarmult_ristretto255_base(gr, r);
crypto_core_ristretto255_add(a, px, gr);

// ----- Second party ----- Send g^k and a^k
unsigned char k[crypto_core_ristretto255_SCALARBYTES];
randombytes_buf(k, sizeof k);

// Compute v = g^k
unsigned char v[crypto_core_ristretto255_BYTES];
crypto_scalarmult_ristretto255_base(v, k);

// Compute b = a^k
unsigned char b[crypto_core_ristretto255_BYTES];
if (crypto_scalarmult_ristretto255(b, k, a) != 0) {
    return -1;
}

// ----- First party ----- Unblind f(x)
// Compute vir = v^(-r)
unsigned char ir[crypto_core_ristretto255_SCALARBYTES];
unsigned char vir[crypto_core_ristretto255_BYTES];
crypto_core_ristretto255_scalar_negate(ir, r);
crypto_scalarmult_ristretto255(vir, ir, v);

// Compute f(x) = b * v^(-r) = (p(x) * g^r)^k * (g^k)^(-r)
//               = (p(x) * g)^k * g^(-k) = p(x)^k
unsigned char fx[crypto_core_ristretto255_BYTES];
crypto_core_ristretto255_add(fx, b, vir);

```

Encoded element validation

```
int crypto_core_ristretto255_is_valid_point(const unsigned char *p);
```

The `crypto_core_ristretto255_is_valid_point()` function checks that `p` is a valid ristretto255-encoded element.

This operation only checks that `p` is in canonical form.

The function returns `1` on success, and `0` if the checks didn't pass.

Random group element

```
void crypto_core_ristretto255_random(unsigned char *p);
```

Fills `p` with the representation of a random group element.

Hash-to-group

```
int crypto_core_ristretto255_from_hash(unsigned char *p, const unsigned char *r);
```

The `crypto_core_ristretto255_from_hash()` function maps a 64 bytes vector `r` (usually the output of a hash function) to a group element, and stores its representation into `p`.

Scalar multiplication

```
int crypto_scalarmult_ristretto255(unsigned char *q, const unsigned char *n,
                                   const unsigned char *p);
```

The `crypto_scalarmult_ristretto255()` function multiplies an element represented by `p` by a scalar `n` (in the `[0..L]` range) and puts the resulting element into `q`.

`q` should not be used as a shared key prior to hashing.

The function returns `0` on success, or `-1` if `p` is the zero element.

```
int crypto_scalarmult_ristretto255_base(unsigned char *q, const unsigned char *n);
```

The `crypto_scalarmult_ristretto255_base()` function multiplies the generator by a scalar `n` (`[0..L]` range) and puts the resulting element into `q`.

The function returns `-1` if `n` is `0`, and `0` otherwise.

Element addition/substraction

```
int crypto_core_ristretto255_add(unsigned char *r,
                                const unsigned char *p, const unsigned char *q);
```

The `crypto_core_ristretto255_add()` function adds the element represented by `p` to the element `q` and stores the resulting element into `r`.

The function returns `0` on success, or `-1` if `p` and/or `q` are not valid encoded elements.

```
int crypto_core_ristretto255_sub(unsigned char *r,
                                const unsigned char *p, const unsigned char *q);
```

The `crypto_core_ristretto255_sub()` function subtracts the element represented by `p` to the element `q` and stores the resulting element into `r`.

The function returns `0` on success, or `-1` if `p` and/or `q` are not valid encoded elements.

Scalar arithmetic over L

Scalars should ideally be randomly chosen in the `[0..L[` interval, `L` being the order of the group ($2^{252} + 2774231777372353535851937790883648493$).

This can be achieved with the following function:

```
void crypto_core_ristretto255_scalar_random(unsigned char *r);
```

`crypto_core_ristretto255_scalar_random()` fills `r` with a `crypto_core_ristretto255_SCALARBYTES` bytes representation of the scalar in the `[0..L[` interval.

A scalar in the `[0..L[` interval can also be obtained by reducing a possibly larger value:

```
void crypto_core_ristretto255_scalar_reduce(unsigned char *r, const unsigned char *s);
```

The `crypto_core_ristretto255_scalar_reduce()` function reduces `s` to `s mod L` and puts the `crypto_core_ristretto255_SCALARBYTES` integer into `r`.

Note that `s` is much larger than `r` (64 bytes vs 32 bytes). Bits of `s` can be left to `0`, but the interval `s` is sampled from should be at least 317 bits to ensure almost uniformity of `r` over `L`.

```
int crypto_core_ristretto255_scalar_invert(unsigned char *recip, const unsigned char *s);
```

The `crypto_core_ristretto255_scalar_invert()` function computes the multiplicative inverse of `s` over `L`, and puts it into `recip`.

```
void crypto_core_ristretto255_scalar_negate(unsigned char *neg, const unsigned char *s);
```

The `crypto_core_ristretto255_scalar_negate()` function returns `neg` so that $s + \text{neg} = 0 \pmod{L}$.

```
void crypto_core_ristretto255_scalar_complement(unsigned char *comp, const unsigned char *s);
```

The `crypto_core_ristretto255_scalar_complement()` function returns `comp` so that $s + \text{comp} = 1 \pmod{L}$.

```
void crypto_core_ristretto255_scalar_add(unsigned char *z,
                                         const unsigned char *x, const unsigned char *y);
```

The `crypto_core_ristretto255_scalar_add()` function stores $x + y \pmod{L}$ into `z`.

```
void crypto_core_ristretto255_scalar_sub(unsigned char *z,
                                         const unsigned char *x, const unsigned char *y);
```

The `crypto_core_ristretto255_scalar_sub()` function stores $x - y \pmod{L}$ into `z`.

```
void crypto_core_ristretto255_scalar_mul(unsigned char *z,
                                         const unsigned char *x, const unsigned char *y);
```

The `crypto_core_ristretto255_scalar_mul()` function stores $x * y \pmod{L}$ into `z`.

Constants

- `crypto_scalarmult_ristretto255_BYTES`
- `crypto_scalarmult_ristretto255_SCALARBYTES`

- `crypto_core_ristretto255_BYTES`
- `crypto_core_ristretto255_HASHBYTES`
- `crypto_core_ristretto255_SCALARBYTES`
- `crypto_core_ristretto255_NONREDUCEDSCALARBYTES`

Algorithms

- `ristretto255`

Reference

- [Ristretto](#)
- [Decaf: Eliminating cofactors through point compression](#)

Defining a custom random number generator

On Unix-based systems and on Windows, Sodium uses the facilities provided by the operating system when generating random numbers is required.

Other operating systems do not support `/dev/urandom` or it might not be suitable for cryptographic applications. These systems might provide a different way to gather random numbers.

And, on embedded operating systems, even if the system may not have such a facility, a hardware-based random number generator might be available.

In addition, reproducible results instead of unpredictable ones may be required in a testing environment.

For all these scenarios, Sodium provides a way to replace the default implementations generating random numbers.

Usage

```
typedef struct randombytes_implementation {
    const char *(*implementation_name)(void);
    uint32_t    (*random)(void);
    void        (*stir)(void);
    uint32_t    (*uniform)(const uint32_t upper_bound);
    void        (*buf)(void * const buf, const size_t size);
    int         (*close)(void);
} randombytes_implementation;

int randombytes_set_implementation(randombytes_implementation *impl);
```

The `randombytes_set_implementation()` function defines the set of functions required by the `randombytes_*` interface.

This function should only be called once, before `sodium_init()` .

Example

Sodium ships with a sample alternative `randombytes` implementation based on the Salsa20 stream cipher in `randombytes_salsa20_random.c` file.

This implementation only requires access to `/dev/urandom` or `/dev/random` (or to `RtlGenRandom()` on Windows) once, during `sodium_init()`.

It might be used instead of the default implementations in order to avoid system calls when random numbers are required.

It might also be used if a non-blocking random device is not available or not safe, but blocking would only be acceptable at initialization time.

It can be enabled with:

```
randombytes_set_implementation(&randombytes_salsa20_implementation);
```

Before calling `sodium_init()`.

It does fast key erasure. However, it is not thread-safe (locks must be added if this is a requirement), and was designed to be just a boilerplate for writing implementations for embedded operating systems. `randombytes_stir()` also has to be called to rekey the generator after `fork()`ing.

If you are using Windows or a modern Unix-based system, you should stick to the default implementations.

Notes

Internally, all the functions requiring random numbers use the `randombytes_*` interface.

Replacing the default implementations will affect explicit calls to `randombytes_*` functions as well as functions generating keys and nonces.

Since version 1.0.3, custom RNGs don't need to provide `randombytes_stir()` nor `randombytes_close()` if they are not required (for example if the data comes from a system call). These can be `NULL` pointers instead. `randombytes_uniform()` doesn't have to be defined either: a default implementation will be used if a `NULL` pointer is given.

Internals

Naming conventions

Sodium follows the NaCl naming conventions.

Each operation defines functions and macros in a dedicated `crypto_operation` namespace. For example, the "hash" operation defines:

- A description of the underlying primitive: `crypto_hash_PRIMITIVE`
- Constants, such as key and output lengths: `crypto_hash_BYTES`
- For each constant, a function returning the same value. The name is identical to the constant, but all lowercase: `crypto_hash_bytes(void)`
- A set of functions with the same prefix, or being identical to the prefix: `crypto_hash()`

Low-level APIs are defined in the `crypto_operation_primitivename` namespace. For example, specific hash functions and their related macros are defined in the `crypto_hash_sha256`, `crypto_hash_sha512` and `crypto_hash_sha512256` namespaces.

To guarantee forward compatibility, specific implementations are intentionally not directly accessible. The library is responsible for choosing the best working implementation at runtime.

For compatibility with NaCl, sizes of messages and ciphertexts are given as `unsigned long long` values. Other values representing the size of an object in memory use the standard `size_t` type.

Avoiding type confusion

An object type has only one public representation.

In particular, points and scalars are always accepted and returned as a fixed-size, compressed, portable and serializable bit string.

This simplifies usage and mitigates type confusion in languages that don't enforce strict type safety.

Thread safety

Initializing the random number generator is the only operation that requires an internal lock.

`sodium_init()` should be called before any other functions. It picks the best implementations for the current platform, initializes the random number generator and generates the canary for guarded heap allocations.

On POSIX systems, everything in libsodium is guaranteed to always be thread-safe.

Heap allocations

Cryptographic operations in Sodium never allocate memory on the heap (`malloc` , `calloc` , etc) with the obvious exceptions of `crypto_pwhash` and `sodium_malloc` .

Prepended zeros

For some operations, the traditional NaCl API requires extra zero bytes (`*_ZERBYTES` , `*_BOXZERBYTES`) before messages and ciphertexts.

However, this proved to be error-prone.

For this reason, functions whose input requires transformations before they can be used are discouraged in Sodium.

When NaCl API compatibility is a requirement, alternative functions that do not require extra steps are available and recommended.

Branches

Secrets are always compared in constant time using `sodium_memcmp()` or `crypto_verify_(16|32|64)()` .

Alignment and endianness

All operations work on big endian and little endian systems, and do not require pointers to be aligned.

C macros

C header files cannot be used in other programming languages.

For this reason, none of the documented functions are macros hiding the actual symbols.

Security first

When a balance is required, extra safety measures have a higher priority than speed.

Examples include:

- Sensitive data are wiped from memory when the cost remains reasonable compared to the cost of the actual computations.
- Signatures use different code paths for verification in order to mitigate fault attacks, and check for small order nonces.
- X25519 checks for weak public keys.
- Heap memory allocations ensure that pages are not swapped and cannot be shared with other processes.
- The code is optimized for clarity, not for the number of lines of code. With the exception of trivial inlined functions (such as helpers for unaligned memory access), implementations are self-contained.
- The default compiler flags use a conservative optimisation level, with extra code to check for stack overflows, and with some potentially dangerous optimisations disabled. The `--enable-opt` switch remains available for more aggressive optimisations.
- A complete, safe and consistent API is favored over compact code. Redundancy of trivial functions is acceptable to improve clarity and prevent potential bugs in applications. For example, every operation gets a dedicated `_keygen()` function.
- The default PRG doesn't implement something complicated and potentially insecure in userland to save CPU cycles. It is fast enough for most applications while being guaranteed to be thread-safe and fork-safe in all cases. If thread safety is not required, a faster, yet intentionally very simple and provably secure userland implementation is provided.
- The code includes many internal consistency checks, and will defensively `abort()` if something unusual is ever detected. This requires a few extra checks, but we believe that they are useful to spot internal or application-specific bugs that tests didn't catch.

Testing

Unit testing

The test suite covers all the functions, symbols and macros of a library built with `--enable-minimal`.

In addition to fixed test vectors, all functions include non-deterministic tests, using variable-length, random data.

Non-scalar parameters are stored into a region allocated with `sodium_malloc()` whenever possible. This immediately detects out-of-bounds accesses, including reads. The base address is also not guaranteed to be aligned, which helps detect mishandling of unaligned data.

The Makefile for the test suite also includes a `check-valgrind` target, that checks that the whole suite passes with the Valgrind's memcheck, helgrind, drd and sgcheck modules.

Static analysis

Continuous static analysis of the Sodium source code is provided by Coverity and Facebook's Infer.

On Windows, static analysis is done using Visual Studio and Viva64 PVS-Studio.

The Clang static analyzer is also used on OSX and Linux.

Releases are never shipped until all these tools report zero defects.

Dynamic analysis

Continuous Integration is provided by [Travis](#) for Linux/x86_64, and by [AppVeyor](#) for the Visual Studio builds.

In addition, the test suite has to always pass on the following environments. libsodium is manually validated on all of these before every release, as well as before merging a new change to the `stable` branch.

- asmjs/V8 (node + in-browser), asmjs/SpiderMonkey, asmjs/JavaScriptCore, asmjs/ChakraCore
- webassembly/V8, webassembly/Firefox, webassembly/WASI
- OpenBSD-current/x86_64 using `clang`
- Ubuntu/x86_64 using gcc 9, `-fsanitize=address,undefined` and Valgrind (memcheck, helgrind, drd and sgcheck)
- Ubuntu/x86_64 using clang 9, `-fsanitize=address,undefined` and Valgrind (memcheck, helgrind, drd and sgcheck)
- Ubuntu/x86_64 using tcc
- Ubuntu/x86_64 using CompCert
- macOS using Xcode 11.2.1
- Windows 10 using Visual Studio 2010 (x86_64 only), 2012, 2013, 2015, 2017 and 2019 (x86 and x86_64)

- `msys2` using `mingw32` and `mingw64`
- `ArchLinux/x86_64`
- `ArchLinux/armv6`
- `Debian/x86`
- `Debian/sparc`
- `Debian/ppc`
- `Raspbian/Cortex-A53`
- `iOS/A12 (iSH)`
- `Ubuntu/aarch64` - Courtesy of the GCC compile farm project
- `Fedora/ppc64` - Courtesy of the GCC compile farm project
- `AIX 7.1/ppc64` - Courtesy of the GCC compile farm project
- `Debian/mips64` - Courtesy of the GCC compile farm project

Cross-implementation testing

(in progress)

[crypto test vectors](#) aims at generating large collections of test vectors for cryptographic primitives, produced by multiple implementations.

[libsodium validation](#) verifies that the output of libsodium's implementations are matching these test vectors. Each release has to pass all these tests on the platforms listed above.

Bindings for other languages

Bindings are essential to the libsodium ecosystem. It is expected that:

- New versions of libsodium will be installed along with bindings written before these libsodium versions were available.
- Recent versions of these bindings will be installed along with older versions of libsodium (e.g. stock package from a Linux distribution).

For these reasons, ABI stability is critical:

- Symbols must not be removed from non-minimal builds without changing the major version of the library. Symbols must not be replaced with macros either.
- However, symbols that will eventually be removed can be tagged with GCC's `deprecated` attribute. They can also be removed from minimal builds.
- A data structure must be considered opaque from an application perspective, and a structure size cannot change if that size was previously exposed as a constant. Structures whose size are subject to changes must only expose their size through a function.

Any major change to the library should be tested for compatibility with popular bindings, especially those recompiling a copy of the library.

Roadmap

libsodium's roadmap is driven by its user community and new ideas are always welcome.

New features will be gladly implemented provided that they are not redundant and solve common problems.

pre-1.0.0 roadmap

- AEAD construction (ChaCha20Poly1305)
- API to set initial counter value in ChaCha20/Salsa20
- Big-endian compatibility
- BLAKE2
- ChaCha20
- Constant-time comparison
- Cross-compilation support
- Detached authentication for `crypto_box()` and `crypto_secretbox()`
- Detached signatures
- Deterministic key generation for `crypto_box()`
- Deterministic key generation for `crypto_sign()`
- Documentation
- Ed25519 signatures
- Emscripten support
- FP rounding mode independent poly1305 implementation
- Faster portable curve25519 implementation
- Fix undefined behaviors for C99
- Guarded memory
- HMAC-SHA512, HMAC-SHA256
- Hex codec
- Hide specific implementations, expose wrappers
- Higher-level API for `crypto_box`
- Higher-level API for `crypto_secretbox`
- Lift `ZEROBYTES` requirements
- Make all constants accessible via public functions
- MingW port
- Minimal build mode
- NuGet packages
- Password hashing

- Pluggable random number generator
- Portable memory locking
- Position-independent code
- Replace the build system with autotools/libtool
- Runtime CPU features detection
- Secure memory zeroing
- Seed and public key extraction from an ed25519 secret key
- SipHash
- Streaming support for hashing and authentication
- Streaming support for one-time authentication
- Support for arbitrary HMAC key lengths
- Support for architectures requiring strict alignment
- Visual Studio port
- 100% code coverage, static and dynamic analysis
- `arc4random*()` compatible API
- Ed25519 to X25519 keys conversion
- iOS/Android compatibility

1.0.x roadmap

- Constant-time `bin2hex()` [DONE] and `hex2bin()` [DONE]
- Constant-time base64 codecs [DONE]
- Improve consistency and clarity of function prototypes
- Improve the documentation
- Consider `getrandom(2)` [DONE]
- Consider [Gitian](#)
- Complete the sodium-validation project
- Optimized implementations for ARM w/NEON
- AVX optimized Curve25519 [DONE]
- Precomputed interface for `crypto_box_easy()` [DONE]
- First-class support for Javascript [DONE]
- chacha20 and chacha20poly1305 with a 96 bit nonce and a 32 bit counter [DONE]
- IETF-compatible chacha20poly1305 implementation [DONE]
- SSE-optimized BLAKE2b implementation [DONE]
- AES-GCM [DONE]
- AES-GCM detached mode [DONE]
- Use Montgomery reduction for GHASH
- ChaCha20-Poly1305 detached mode [DONE]
- Argon2i as `crypto_pwhash` [DONE]

- Argon2id as `crypto_pwhash` [DONE]
- Multithreaded `crypto_pwhash` [on hold]
- Generic subkey derivation API [DONE]
- Nonce-misuse resistant scheme
- BLAKE2 AVX2 implementations [DONE]
- Keyed (hash-then-encrypt) `crypto_pwhash`
- Consider Yescrypt
- Consider BLAKE2X [on hold]
- Argon2id [DONE]
- Port libhydrogen's key exchange API
- SSSE3 ChaCha20 implementation [DONE]
- SSSE3 Salsa20 implementation [DONE]
- SSSE3 Poly1305 implementation [DONE]
- AVX2 Salsa20 implementation [DONE]
- AVX2 ChaCha20 implementation [DONE]
- AVX2 Poly1305 implementation
- AVX512 implementations [done for Argon2, withhold for other operations due to throttling concerns]
- key generation API [DONE]
- Nonce/subkey generation API
- Webassembly support [DONE]
- Stream encryption using a CHAIN-like construction [DONE]
- Security audit by a 3rd party [DONE]
- Formally-verified implementations [on hold]
- Padding API [DONE]
- `secretstream_inject()` for nonce misuse-resistance [on hold]
- Point addition, subtraction [DONE]
- Point validation [DONE]
- Hash-to-point (Elligator) [DONE]
- SPAKE2+ [DONE]
- Support server relief in the password hashing API
- Ristretto [DONE]
- Consider a streaming interface for `crypto_shorthash_*`
- AEGIS-256 [PARTIALLY DONE]
- BLAKE3