Sodium

crypto library

Table of Contents

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 supports a variety of compilers and operating systems, including Windows (with MinGW or Visual Studio, x86 and x86_64), iOS and Android.

The design choices emphasize security, and "magic constants" have clear rationales.

And despite the emphasis on high security, primitives are faster across-the-board than most implementations of the NIST standards.

Version 1.0.10 was released on Apr 5, 2016.

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 ePUB (for iPad, iPhone, Mac), MOBI (for Kindle) and PDF here: https://www.gitbook.com/book/jedisct1/libsodium/details

License

ISC license.

See the `LICENSE` file for details.

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, then follow the ritual:

```
$ ./configure
$ make && make check
# make install
```

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 and 2015 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 `__dllexport`.

Cross-compiling

Cross-compilation is fully supported. 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.

Compiling with CompCert

Releases can be compiled using the CompCert compiler. However, when using CompCert, the Autoconf scripts might mistakenly detect byte ordering as big endian even on little endian systems.

You may want to manually predefine `ac_cv_c_bigendian=no` in order work around this.

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

```
env CC=ccomp ac_cv_c_bigendian=no \
    CFLAGS="-O2 -fstruct-passing" ./configure \
    --disable-shared --enable-static && \
make check && 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.

Integrity checking

Distribution files can be verified with Minisign and the following Ed25519 key:

```
RWQf6LRCGA9i53mlYecO4IzT51TGPpvWucNSCh1CBM0QTaLn73Y7GFO3
```

Or with GnuPG and the following RSA key:


```
-----BEGIN PGP PUBLIC KEY BLOCK-----
Version: GnuPG v1 (OpenBSD)

mQINBFTZ0A8BEAD2/BeYhJpEJDADNuOz5EO8E0SIj5VeQdb9WLh6tBe37KrJJy7+
FBFnsd/ahfsqoLmr/IUE3+ZejNJ6QVozUKUAbds1LnKh8ejX/QegMrtgb+F2Zs83
8ju4k6GtWquW5OmiG7+b5t8R/oHlPs/1nHbk7jkQqLkYAYswRmKld1rqrrLFV8fH
SAsnTkgeNxpX8W4MJR22yEwxb/k9grQTxnKHHkjJInoP6VnGRR+wmXL/7xeyUg6r
EVmTaqEoZA2LiSaxaJ1c8+5c7oJ3zSBUveJA587KsCp56xUKcwm2IFJnC34WiBDn
KOLB7lNxIT3BnnzabF2m+5602qWRbyMME2YZmcISQzjiVKt8O62qmKfFr5u9B8Tx
iYpSOal9HvZqih8C7u/SKeGzbONfbmmJgFuA15LVwt7I5Xx7565+kWeoDgKPlfrL
7zPrCQqS1a75MB+W/fOHhCRJ3IqFc+dT1F4hb8AAKWrERVq27LEJzmxXH36kMbB+
eQg336JlS6TmqelVFb15PgtcFh972jJK8u/vpHY0EBPij5chjYQ2nCBmFLT5O4UZ
Y4Gm8Z3QLFG1EeOiz+uRdNfchxwfLkjng1UhKXSq5yuOAAeMaNoYFtCf1hAHG6tx
vWyIijRxUd5c8cDZsKMuLQ34O6DuvPZyeCy6q8BTfW18miMMhIH0QTS9MwARAQAB
tC5GcmFuayBEZW5pcyAoSmVkaS9TZWN0b3IgT25lKSA8akBwdXJlZnRwZC5vcmc+
iQI2BBMBCAAgAhsDAh4BAheABQJU2dF6BAsJCAcFFQoJCAsFFgIDAQACgkQIQYn
qrpwn+GpOBAAkJu5yZhLPBIznDZMr0oJ/pJiSea7GUCY4fVuFUKLpLlSjIaSxC4E
2oWG8cJoMdMhwW1x166rRZPdXFpW8eC5r+h8m25HBJ649FjMUPDi2r9uQgPdBy80
I+gFlrsinSy7xbdlUSpjrcYYCx9jYjjTwH6L1QZa+YCMFya8dob/NcdzQ0o7cNRu
5NG988cScsscXYXzI6SMouSwPGCMrQHAsM31Yb8YFbJLuDxFRCZY5+qiR8DXDzW4
Lp68fJq0X/UGW9Q+i29LMTvZZWDGBQ9bwQNtvDrPZ8SYp249cMOsR4W7FK4Y0Oea
YRTBFcXaeXEKAP1ZqYrY22BDiHJO5IGY72D3j3vPATAYigwjr/qNFOt/DaERFpQ4
L7RD+E6WLHATFWxZHH/APck6q8bY4EHr8GJWA77sIqN/Ctvap759QKB8nrerT6lA
0cojhS5Ie8Lro6YsMAXDqwjzsv+VgnTgql8oAFmuU+o+6cmHUwGNHgEs+xe2UDQi
kxu685gOCHfHmBwue391glHufQdveChy5eikif6q6Ndg7VH9mR335o8VJ9I+Vp/k
3W8XZBA9OEuwrxjy1EzWvcb2WGXrUHVZ32w+E9CICvFFV7JiTntG3t1Ch4/bbFwr
wdkc5EZTh0c6B7YfIkEWnOnBovWBPEBkSGve371MsqBuKuBr1W4jecyIRgQQEQgA
BgUCVNnRHAAKCRCSa8UXHN6kOWXzAKCGlk6DvVCqExkBd6OEsaEoOBgH5ACfcVQa
z/FEgCdRsJeLi7xNwZXZ22O0IUZyYW5rIERlbmlzIDxnaXRodWJAcHVyZWZ0cGQu
b3JnPokCNgQTAQgAIAIbAwIeAQIXgAUCVNnRaQQLCQgHBRUKCQgLBRYCAwEAAAoJ
ECEGJ6q6cJ/hslIQAI2l+uRlwmofiSHo/H2cUDNO2Nn7uRfcVIw9EItTmdU6KKx9
nkgFP3Y3lUwkLQFP6aQhQJyHBU5QGqn9n8k8+jEPciTL7hcbTuY0YRuz0mp9bJ8r
ruqGxTrZuogvIVntwnn1HvgAbu13HKu+3KOLYDmWqosVNf0a8GjHj10ZDuNDPQVb
```

```
X6NWDes+jLdeUsxVKUZHlOC3CiRCSHJzZ3G1gO9QU78LQAFCIIDO7GO7xPjqbvEX
nsys5f12OLXB4NqBlIamEdyztV+CwIZBM9Ni6ytPnEhWzTHzHwi95oNa+AtpUlgG
RYjYtMR9pxCqVkrplwrwhA4dbSO7HLiXQIrA57F1/5LwKRR4e7IGhnTpZoW8hr8y
qg4AAVCZqr5aB82LOJAMP6ZlC7kQb9/YxGYw4Vwf6qCY8Iw74MvIL5wW0zSv/orB
eNtHeP0Z/Ozx3UXKA2chNElEWbZ9e0IZBXgcj/JDfK8e0VTqv1ItHLm2ZkvCbyhV
fER8I8AHPnfzwkXvWFeDKeMO8rakqDeNQ3h4BeiCBCVHpEsUdIWSG3oCO1guy9/h
xMJR2yAWiK+35sCcZbrgTTN0oQepRMuZ34niIBK0jUh7t1M5sBMNgxEAIeKjJf64
DEudNz+xUgek5N+BXx7hryuVC3s1y6H42ztOjPtpHPVUw98gWpv5V7QRLBs0iEYE
EBEIAAYFAlTZ0RwACgkQkmvFFxzepDn8sACdF51BycwRvMpkFPea1Yi3/B1EOs0A
oJT9afe3zQnOlcIuBFBzpdOTsecUtCZGcmFuayBEZW5pcyA8ZnJhbmsuZGVuaXNA
Y29ycC5vdmguY29tPokCNgQTAQgAIAIbAwIeAQIXgAUCVNnRegQLCQgHBRUKCQgL
BRYCAwEAAAoJECEGJ6q6cJ/h0LgP+wfCw2SCFvD7sFlnmd6oJNP+ddtt+qbxDGXo
UbhrS1N88k6YiFRZQ+Z84ge9RgQXA74xuWlx8g1YBEsqO1rYCGQ4C+Ph+oUO+a3X
k+wmEzINnjCF8CQzZQ3vdXvWmshKzqC2yyeR235WC/BSHsqsr+TRFEmGa68ju8s7
UF8ZQaBzbM0ttUtrc0UqhnS16xV5lH9gBkVbMWIN1pAeJcFRL6MB92Vv5tWjayua
w76vxmwPhu6quUlwxNYNvYBgG5kpBjqMOLHaX1x+SA5F6aI6E3kqxeyurwV6Ty+/
FIns+Awl+IFPey5ctwSOXkizhtqxpMNHAu9resNRjneIjNVTLON1uaxvmPJttMd/
CdTXh+guxDBfH6Vr9nmExy2qbihDJ06Sm874UYtnBZdB7Fi0cNF1DlEZKaZyYaLw
RA/TelI2IaIdkRFLsaFdo144nfceZ2fra2QO83Ow6uShNZzAHU0ZVEKLVt/VJqCL
6hts7vhKuCBcNlpoNOZptRPJf8RMLh4qwtniZadDcM16TpvkyTQUAWH+GvTML0UR
5sLHOtZ7MUaHO/c5UWQWJOmuaWOKgdKLi3iXztGbNNDc9F7wRoObUH7Om/0s5IRy
noO58ofDCmurPDP+10eOQaWtgVz2nFXcFF0qTw4H6L/sXlzbm27HuqEHuYrzpTl/
Njn0chjBiEYEEBEIAAYFAlTZ0RwACgkQkmvFFxzepDnrmQCfdaiJcQsAZaSfEfO1
VxZaY0kEVf0An1xVULYvo5M4sta0tILFu3UthzBGtDdGcmFuayBEZW5pcyAoSmVk
aS9TZWN0b3IgT25lKSA8MGRheWRpZ2VzdEBwdXJlZnRwZC5vcmc+iQI2BBMBCAAg
BQJU2dKRAhsDBAsJCAcFFQoJCAsFFgIDAQACHgECF4AACgkQIQYnqrpwn+FqRxAA
wWm+f6mo9nCoGRD4r4jrSLuJ5ApyIxRQ3L5DL/MeITRMPNDps0OpvKIIGmGv19n5
Ani7ufOcnQLkTVj1179U5BTnahk2fDS0CxlFyslpR9A7tX6qQMtIyBE4cdPhjVue
ZOwI+PfJSleFFmPQ3ESlbKzeNGJqBQiNSbpo9qMhhyYRZy/Fk4kOQzAdXpa63kPX
1KVoTsvz19O2frLim7QY8oTI8Vbij0CB+HfhHuLmolc039/S47hF+5ygERK5Fwjo
mSx+Q2fKx9P35TZqQ9Zw73e3gS9YUErT4LU7ZwdmulftfCaVLmIuX4GUDPasmNbA
WLpKHEwLln0YJO0kIzD+2q2zclzUmGgdgGcEUwLb6vpWLJ41MsmHknZg0zm/yG6/
sasA0jU1wKxeRlHeSxnh3PYb+v36kHXsRViqPlwxe9PGmLK9p9wD0yS/dk2LsJbE
1hnUZfw7l14VdivrL567My/0sG3SbIUb/DxHuVkgHU9LHHlca4z5VmFc7v2+sc0+
6IczFW86FKI8m+q8zLhHcquKgZpumxvwjEoAbjl9123bqZKm1e8pHL3bTQa6bSv9
isNsW3T9eHeEB7frbBlYOZjvMQuYLf82t2tu+E4xbUYZZrmlRYGwBGFUBRprtJ0e
XeUvxFgAnazyNNXxXhO3PMiCxpCp0e7+x64fKVPMfFu5Ag0EVNnQDwEQAMnv/UG9
7vAtIyeG+lPalmhn10NQ07I4Rz+vigZHAxO8t7QYhOYOYLZFj1mO11f8lc5X1oxV
7dKwh+sHMJQ3fkOmQbG6VGRLmRTAPk45GsaRcAnczNzCZWw0s4f92ybc9Th4dNR8
dUk90t+tFItPGnFHGHmjwUYMc7u8BNl9l/SNiJipxuHjUR1hXQE+RXrlgkoW9S8I
bisHytd5IcOXGz337coYkdJLzx1AdpOMGN4n5qymlrhjBIvV2a/R+mweUAD7Il8I
Ynj58lalrp2kLmnoJacL0R9R2ZbSjDBevKpitmy3kqHS59vChw80asBRWr10++Ea
V0LnWDKKbc1U809RP1Ac0l66KjKj3mmiQQKDpb2oHHD0uJsx84kqCOkoWdqF12wR
stygYsAc8CJXnsAKThdDvsQTkMX6WKg4wtSJw0ELRtNCQZzH8iE6eq9MXZijvG6H
j9WyZ2L2eeO0bKn0uEDGvpPMLWcFfOjCxL32x/Jr95sqAt2p0DcBFH5d4jK7tqHQ
YzNwt8ibbbGlwzRFTgq/5igV+n9q9P/h8bWQhUJyqbjyJuwt4l/oTSTKZ5bZ0IAr
KS/+Y/Y9b/BBXRzRP/D1LhaOndH43E6HmEWGS2PhUUPn3V6TQzOq5npaTXKhq/f8
XMYEqvbQ3qjfREa+LLgmFLAwD7rc8h2WYVp7ABEBAAGJAh8EGAEIAAkFAlTZ0A8C
GwwACgkQIQYnqrpwn+GCVhAAscO0pYCRzcgDwDWOrT3g5yi8dt3NmDGL9c6/ohKV
waWSIDlwFtbZNiZ/fr91VCdDfhUSohtn6E7XvKYdVNO4NRLIbSgRc7Y/C4P+9lEh
k+6mlXYlEil/GN6YXBsQvDSz1xw+Csz3Y6kq2m1xiSHFuZrP0PS75x+vIAKbIspa
uu5IyEh/wAW1vY/pnzs7TJtY2r8Qsv/5xt+zUdlGB0ZJq7IZ/1GveltRMJrfhcCT
KPQRWdMv0aEioeBwYAM8sc9UrrePM9jSpT3uCYwuJlld4M94+tqt7tqvkR6dluXF
```

```
+4WWeuPXo65jSBl094BEfT5dVbt0TqmG6eTgnPghh1j7PpIghyqUU0v8YPl5DUnZ
UuHzi4CEcQWNUEq+xK9N2/nflaq8R4LPDJjupSWIw5tZv8NWj+EA/zyxggX+q2pr
3qlD+IUnO8cR/RT1LvZ9L5t1fvTqjpgDqXJIremihObLOGEV0+0xWEaN085OVzyU
QTt2EBhzSxHkC0CEd6CgR8l48YGsKJrHCjuOvQ+lgVtAkgYBeVFefhrKa242TmVB
NlZCkS25wUhGhWbLv334p+KTG4d79J+iKYbh8n0C/gBK0YzDX3gLbL+6wes0xYia
WSRBfx9hfPCfFLDGG5sY7yViH8YcOGig6IV9+DWBCSyOZ0d0IXWNvTLF+3d1BFD4
dlG5Ag0EVNnQNwEQANZNoFI4cM9TYFCMOYIiH1UaXoibNE7kZ1qDM/O6y5HTUOSn
m2koCYMTqtVaigAq/tXiUJLBzoHwh17CzDx5L3/IShMHdqwAFCcUZII2NW/XEEH7
knwnqn5tki2CZCzfE+GXtUm7M7fBW2pgPvVt/Ord+DhmEKP0A+fdKHS3x/EUn8Vs
vJoYEkxg9fT14eqYk+oALFxm6vW9UAFO0VZ/JOXzeDTux0+6p6NQjcykKeG5GiXA
dHpRopfeksLQx3sZqfFBEhuiIX7PllAQxHpPqKcPG82aVqT5x9tvZ2RVdk/55hcK
gNhdcbDGWqkNENbOvTmom2a/gDNgb7pf12jJa9t2RRVC8oyYh+zVftLhf2GlwMVv
vwuXO1U2A0/lUQ7K33t6lQ2mEmbudyeFJCso3kIJ598efTw2ZPkeEkZ+adsIBQbd
CSEm0B/S+DS8CDTLTfS5nN5T3rGnO7lzPf983uP9CLbODyt05dqF1Hl+4XicMT3P
Qtz1T+P7X7nPQL9FUwOWUBHqfhYhNsnV17m6M/ODoKsyjdl92njOxvyD6zVaffcx
2zX+SYEaIIiDFhxVFprhwTuruKOfax3nNTLd1JeiraUejSNCnP60VxTsp203Y0H8
quLtvsWF6V5lr57WQxGQxQmS5JQV9wreYzuA339ApUqukfWmhiPDHbQVWAe3ABEB
AAGJBD4EGAEIAAkFAlTZ0DcCGwICKQkQIQYnqrpwn+HBXSAEGQEIAAYFAlTZ0DcA
CgkQYvJbWStvdtq1jg/8Dm6BicjEbcNphWpsjj0uoPB49I0fKFxSM2uUh6PI+wtc
LtikJsNyGvXDm7oGE/uXIki5S++91pZ5oTV931HVzp8e4vip5IRCcWFk6NisRmiZ
nN/xMejLnK3s51pmK5UJhoYymrETGiUKj1uu5BqewRXZ4wWH2kzIusBzIc537shR
Gqk+LgwY7/x4aKY+5Z46VpAGSlO4a6WdWtlRLZzOz0x+tPIrAYo0f72hdHg2enZE
rqkhi90dy/5hCsaJRl+raEZVDSggOtO0hmhTnLSWAX3YPINp1qSqvn5EQk8FhZuh
RaonpXg0wZLc82oIYEZ0KnhJ7HBgV/jF78lI5ZPdk9m22GbASWkIjwNmfzAhGEPu
/NX3iweDPfU4ULbOvejs3ivQTEOrF47u3ps/6SOrBXS7f23ZBw7nwYryezCeQUV8
RCKkk+xUPv5YU0DpGtViDrfxeucXW8W05VOBsCfpa2PTXvj4VjP6UGRUcX3SVTcA
VnvKAmfsDa/4+4AOEvfgQFRzuex8tthFbPW2pLJEQPpVFuxAK0foUHw78HFL7NRV
TFx3jUWgGAM7PA9FI9h1rrU5dXyi8uXwBjaXcEaIts7WE0NGjFzEbub6kJldryhl
5ZCMkmOcBU7SkSmI95bOJwvYdGGiEcO4eh7ci4pOFH0ZNqKfpjyfpTgtFgS5Ldne
pBAA8ubnR6+b7gGaOQk/rROTYHoSq9GXVAqhhmY69lfsXQ9EXoiAzNZnhJLtj1J7
86Z3Bgd9X+MXrrPoJLVGmBTT8yT337KY/+rbk16E5oL1eItnsJ0xgprD1gkWUNaa
pRXLKdA86ogoU8sE/9Wr2CN6dCdPCmjmc0mWvGHY5V6lMf3NPIAQbS4izuU/w+IE
gPnBo45BPkxP2HyvhoOek+pxpsqL8uLQzuIjtwgWvMOocVQrpBNr6kQ99hvr8feY
6kOI5MoGsagW3R65m7DAfz/x1oO3QmWT/kg2dcWqiEbzL3phX1QpQtdJkO5+JTYQ
F0WP5sPzQ7DaIP7Mo2NjhqvnO5NR9/kEzX1yEQck3BI4vKNHSiAQ1/J94uiu9Aze
W6ddPO4Ax7LycK0WOeNVNAT6a3tFJbQrve3ZoDDSNXAa70VKmpdrsrwnX+/4+rly
Z7lj7rnMWCe9jllfZ2Mi+nIYXCrvhVh0t7OHVGwpSq28B/e2AFsQZxXcT4Y+6po7
aJADVdb+LlOAuF6xB3sylE1Im0iADCW9UAWub1oiOr9jv0+mHEYc3kaF0kPU5zKO
I9cg891jcOBV/qRv89ubSHifw9hTZB0dDjXzBjNwNjBHqkYDaLsf1izeYHEG4gEO
sjoMDQMqgw6KyZ++6FgAUGX5I1dBOYLJoonhOH/lNmxjQvc=
=Hkmu
-----END PGP PUBLIC KEY BLOCK-----
```

Applications using libsodium

Some applications using Libsodium. Send a pull request to add yours to that list.

- CRUD Messenger: A libsodium-php and MongoDB based single server messaging platform.
- CurveLock: Message encryption for Windows.
- Cyph: End-to-end encrypted, authenticated, and ephemeral chat with voice + video + file transfers.
- Discord: All-in-one voice and text chat for gamers that's free, secure, and works on both your desktop and phone.
- Evernym Plenum: A byzantine fault tolerant protocol.
- Fastd: A fast and secure tunnelling daemon.
- Glorytun: A small, simple and very fast VPN.
- Hat-Backup: A backend-agnostic snapshotting backup system.
- KadNode: A small P2P DNS resolution daemon based on a Distributed Hash Table.
- Kickpass: A stupid simple password manager.
- Lawncipher: An embedded, encrypted, multi-purpose document store.
- Lageant: Libsodium Authentication Agent.
- Lumimaja: PasswordSafe with Argon2 KDF, data encrypted with ChaCha20Poly1305, and Yubikey support.
- 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.
- Minisign: A dead simple tool to sign files and verify signatures.
- MLVPN: A multi-link VPN (ADSL/SDSL/xDSL/Network aggregator)
- Molch: An implementation of the Axolotl ratchet.
- NanoChat: A P2P, E2E encrypted and discoverable chat application on top of nanomsg library.
- NTPsec: A security-hardened implementation of Network Time Protocol Version 4.
- 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.
- 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.
- 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.
- 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.
- Simple DnsCrypt: A simple management tool for dnscrypt-proxy.
- Splonebox: An open source network assessment tool with focus on modularity.
- Stellar: An open platform for building financial products that connect people everywhere.
- Tbak: Encrypted, compressed, distributed backups.
- Telehash: An embeddable private network stack for mobile, web, and devices.
- Thrussh: A portable SSH client and server library.
- Tinfoil Chat NaCl: TFC-NaCl is a high assurance encryption plugin for the Pidgin IM client.
- Tox: A new kind of instant messenging.
- 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.
- VPNCloud: A Peer-to-Peer VPN.
- Wire: Modern, private communications. Crystal clear voice, video and group chats. No advertising. Your data, always encrypted.
- Zcash: Zcash is a decentralized and open source cryptocurrency that aims to set a new standard for privacy through the use of groundbreaking cryptography.

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.
- Halite: High-level cryptography interface for PHP.
- Libsodium-Laravel: Laravel integration.
- Libsodium-UE4: An easy to use cryptography plugin for Unreal Engine 4 based on libsodium.
- LibSQRL: SQRL 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.
- minisign-net: .NET library to handle and create minisign signatures.
- NaclKeys: Library to generate libsodium-net compatible KeyPair`s.
- 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
- Stream Cryptor: Stream encryption & decryption with libsodium and protobuf for .NET
- The Update Framework: A plug-and-play library for securing a software updater.
- ZeroMQ: Connect your code in any language, on any platform.

Companies using libsodium

Some companies using Libsodium, possibly in non-opensource products. Send a pull request to add yours to that list.

- Digital Ocean
- EAM GmbH (bytejail)
- Informatica
- Keybase
- OVH
- Paragon Initiative Enterprises
- Supercell
- Yandex

Bindings for other languages

- .NET: libsodium-net
- C++: sodiumpp
- C++: tears
- Clojure: caesium
- Clojure: naclj
- Common LISP: cl-sodium
- D: Chloride
- D: Shaker
- D: Sodium
- Delphi/FreePascal: Delphi/FreePascal
- Dylan: libsodium-dylan
- Elixir: Savory
- Erlang: ENaCl
- Erlang: Erlang-NaCl
- Erlang: Erlang-Libsodium
- Erlang: Salt
- Fortran: Fortium
- Go: GoSodium
- Go: libsodium-go
- Go: Natrium
- Go: Sodium
- Haskell: Saltine
- HaXe: haxe_libsodium
- Idris: Idris-Sodium
- Java (Android): Libstodium
- Java (Android): Robosodium
- Java (Android): Libsodium-JNI
- Java: Kalium
- Java: sodium-jni
- 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
- Julia: Sodium.jl
- Lisp (CFFI): foreign-sodium

- Lua: lua-sodium
- MRuby: mruby-libsodium
- Nim: Libsodium.nim
- Nim: Sodium.nim
- OCaml: ocaml-sodium
- Objective-C: NAChloride
- Objective-C: SodiumObjc
- PHP: PHP-Sodium
- PHP: libsodium-php
- Perl: Crypt-Sodium
- Perl: Crypt::Nacl::Sodium
- Pharo/Squeak: Crypto-NaCl
- Pony: Pony-Sodium
- Python: Csodium
- Python: LibNaCl
- Python: PyNaCl
- Python: PySodium
- Q/KDB: Qsalt
- R: Sodium
- Racket: Natrium
- Racket: part of CRESTaceans
- Ruby: RbNaCl
- Ruby: Sodium
- Rust: Sodium Oxide
- Rust: libsodium-sys
- Swift: NaOH
- Swift: Swift-Sodium

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.

`sodium_init()` initializes the library and should be called before any other function provided by Sodium. The function can be called more than once, but it should not be executed by multiple threads simultaneously. Add appropriate locks around the function call if this scenario can happen in your application.

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` is the library had already been initialized.

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 in order 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 should be at least `bin_len * 2 + 1`.

The function returns `hex` on success, or `NULL` on overflow. 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.

The function returns `-1` if more than `bin_maxlen` bytes would be required to store the parsed string. It returns `0` on success and sets `hex_end`, if it is not `NULL`, to a pointer to the character following the last parsed character.

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

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.

This function was introduced in libsodium 1.0.4.

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) mod 2^(8*len)` in constant time for a given length, and overwrites `a` with the result.

This function was introduced in libsodium 1.0.7.

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.

It was introduced in libsodium 1.0.6.

Testing for all zeros

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

This function returns `1` is 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.

This function was introduced in libsodium 1.0.7.

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 senstive 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 neccessary. `sodium_lock()` 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_unlock()` 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.

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. Modification 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 `0xd0` bytes in order to help catch bugs due to initialized 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()`, the only one requiring extra care being `crypto_generichash_state`, whose size needs to be rounded up to a multiple of 64 bytes.

Allocating `0` bytes is a valid operation, and 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.

```c
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.

```c
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 Linux kernels, the `getrandom` system call is used (since Sodium 1.0.3)
- 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 does its best to guarantee a uniform distribution of the possible output values.

```
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.

```
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

(this documentation is a work in progress. Feel free to contribute a nice intro to secret-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 nonce[crypto_secretbox_NONCEBYTES];
unsigned char key[crypto_secretbox_KEYBYTES];
unsigned char ciphertext[CIPHERTEXT_LEN];

randombytes_buf(nonce, sizeof nonce);
randombytes_buf(key, sizeof key);
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/sign 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()` .

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.

Constants

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

Algorithm details

- Encryption: XSalsa20 stream cipher
- Authentication: Poly1305 MAC

Notes

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.

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];

randombytes_buf(key, sizeof 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.

Constants

- `crypto_auth_BYTES`
- `crypto_auth_KEYBYTES`

Algorithm details

- HMAC-SHA512256

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 store protocol-specific metadata about the message, such as its length and encoding.

Supported constructions

Libsodium supports two popular constructions: AES256-GCM and ChaCha20-Poly1305.

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.

If portability is not a concern, AES256-GCM is the fastest option.

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 a faster alternative to the well-studied 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, and is the recommended choice for most applications.

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 two versions of the ChaCha20-Poly1305 construction:

- The original construction can safely encrypt up to 2^64 messages with the same key, without any practical limit to the size of a message (up to 2^70 bytes).
- The IETF variant is slightly slower. It can safely encrypt a pratically unlimited number of messages (2^96), but individual messages cannot exceed 1 terabyte.

Both are interoperable with other crypto libaries, share the same security properties and are accessible via a similar API.

The `crypto_aead_chacha20poly1305_*()` set of functions implements the original construction, while the `crypto_aead_chacha20poly1305_ietf_*()` functions implement the IETF version. The constants are the same, except for the nonce size.

The original ChaCha20-Poly1305 construction

The original ChaCha20-Poly1305 construction can safely encrypt up to 2^64 messages with the same key, without any practical limit to the size of a message (up to 2^70 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;

randombytes_buf(key, sizeof 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 and the encrypted message are stored together. This is usually what you want.

```
int crypto_aead_chacha20poly1305_encrypt(unsigned char *c,
                                         unsigned long long *clen,
                                         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,
                                         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` is 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` .

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 pro tocol, either use different keys for each direction, or make sure that a bit is masked in one direction, and set in the other.

The API conforms to the proposed API for the CAESAR competition.

A high-level `crypto_aead_*()` API is intentionally not defined until the CAESAR competition is over.

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 pratically unlimited number of messages (2^96), but individual messages cannot exceed 1 terabyte.

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;

randombytes_buf(key, sizeof 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 and the encrypted message are stored together. This is usually what you want.

```
int crypto_aead_chacha20poly1305_ietf_encrypt(unsigned char *c,
                                              unsigned long long *clen,
                                              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,
                                              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` is 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` .

Constants

- `crypto_aead_chacha20poly1305_IETF_ABYTES`

Since Sodium 1.0.9:

- `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 pro tocol, either use different keys for each direction, or make sure that a bit is masked in one direction, and set in the other.

The API conforms to the proposed API for the CAESAR competition.

A high-level `crypto_aead_*()` API is intentionally not defined until the CAESAR competition is over.

See also

- ChaCha20 and Poly1305 for IETF protocols - Specification of the IETF variant

Authenticated Encryption with Additional Data using AES-GCM

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 */
}

randombytes_buf(key, sizeof 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 and the encrypted message are stored together. This is usually what you want.

```
int crypto_aead_aes256gcm_encrypt(unsigned char *c,
                                  unsigned long long *clen,
                                  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 `mlen + 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 *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 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 `aead_aes256gcm_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` is 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_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 in 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 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` is 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.

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.

It is recommended to split message larger than 2 Gb into smaller chunks.

Support for AES256-GCM was introduced in Libsodium 1.0.4.

The detached API was introduced in Libsodium 1.0.9.

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.

It is recommended to split message larger than 2 Gb into smaller chunks.

Support for AES256-GCM was introduced in Libsodium 1.0.4.

The detached API was introduced in Libsodium 1.0.9.

Public-key cryptography

(this documentation is a work in progress. Feel free to contribute a nice intro to public-key cryptography!)

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 verify that the encrypted message was actually created by Bob and 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_open_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.

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).

This system provides mutual authentication. However, a typical use case is to secure communications between a server, whose public key is known in advance, and clients connecting anonymously.

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_PUBLICKEYBYTES];
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.

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! */
}
```

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

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

The `crypto_sign_open()` function checks that the signed message `sm` whose length is `smlen` 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 `mlen` if `mlen` 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.

```c
int crypto_sign_detached(unsigned char *sig, unsigned long long *siglen,
                         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.

```c
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.

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).

Constants

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

Algorithm details

- Signature: Ed25519

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.

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 || recip
ient_pk))
```

Availability

`crypto_box_seal` was introduced in Sodium 1.0.3.

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

```c
#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 function 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

```c
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 by 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 reinitializated using `crypto_generichash_init()`.

This alternative API is especially useful to process very large files and data streams.

State structure size

The `crypto_generichash_state` structure length is either 357 or 361 bytes. 64-bytes alignment is recommended for performance, but not required. For dynamically allocated states, `crypto_generichash_statebytes()` returns the rounded up structure size, and should be prefered 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];

randombytes_buf(key, sizeof 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 `randombytes_buf()` .

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

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 leverages the Argon2 function.

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.

Argon2 has two variants: Argon2d and Argon2i. Argon2i uses data-independent memory access, which is preferred for password hashing and password-based key derivation. Argon2i also makes multiple passes over the memory to protect from tradeoff attacks.

This is the variant implemented in Sodium since version 1.0.9.

Argon2 is recommended over Scrypt if requiring libsodium >= 1.0.9 is not a concern.

Scrypt

Scrypt 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.

Scrypt is available in libsodium since version 0.5.0, which makes it a better choice than Argon2 if compatibility with older libsodium versions is a concern.

The Argon2 memory-hard function

Since version 1.0.9, Sodium provides a password hashing scheme called Argon2.

Argon2 summarizes the state of the art in the design of memory-hard functions.

It aims at the highest memory filling rate and effective use of multiple computing units, while still providing defense against tradeoff attacks.

It prevents ASICs from having a significant advantage over software implementations.

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. `outlen` should be at least `16` (128 bits).

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.

`memlimit` is the maximum amount of RAM that the function will use, in bytes.

`alg` is an identifier for the algorithm to use and should be currently set to `crypto_pwhash_ALG_DEFAULT`.

For interactive, online operations, `crypto_pwhash_OPSLIMIT_INTERACTIVE` and `crypto_pwhash_MEMLIMIT_INTERACTIVE` provide base line for these two parameters. This requires 32 Mb 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 128 Mb 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 512 Mb 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 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 the password `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.

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_DEFAULT`
- `crypto_pwhash_SALTBYTES`

- `crypto_pwhash_STRBYTES`
- `crypto_pwhash_STRPREFIX`
- `crypto_pwhash_OPSLIMIT_INTERACTIVE`
- `crypto_pwhash_MEMLIMIT_INTERACTIVE`
- `crypto_pwhash_OPSLIMIT_MODERATE`
- `crypto_pwhash_MEMLIMIT_MODERATE`
- `crypto_pwhash_OPSLIMIT_SENSITIVE`
- `crypto_pwhash_MEMLIMIT_SENSITIVE`

Notes

`opslimit`, the number of passes, has to be at least `3`. `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. Save the parameters along with the hash instead, and use these saved parameters for the verification.

Alternatively, use `crypto_pwhash_str()` and `crypto_pwhash_str_verify()`, that automatically take care of including and extracting the 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 doesn't have to be done before calling this function.

Algorithm details

- Argon2i v1.3

The Scrypt memory-hard function

As a conservative alternative to Argon2, Sodium provides an implementation of the Scrypt password hashing function.

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`.

`opslimit` represents a maximum amount of computations to perform. Raising this number will make the function require more CPU cycles to compute a key.

`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.

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_scrypts
alsa208sha256_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_SALTBYTES`
- `crypto_pwhash_scryptsalsa208sha256_STRBYTES`
- `crypto_pwhash_scryptsalsa208sha256_STRPREFIX`
- `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. Save the parameters along with the hash instead, and use these saved parameters for the verification.

Alternatively, use `crypto_pwhash_scryptsalsa208sha256_str()` and `crypto_pwhash_scryptsalsa208sha256_str_verify()` , that automatically take care of including and extracting the 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);
```

Diffie-Hellman function

Sodium provides X25519, a state-of-the-art Diffie-Hellman function suitable for a wide variety of applications.

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 (≈2^252), and the key distribution is not uniform. For this reason, instead of directly using the output of the multiplication `q` as a shared key, it is recommended to use `h(q || pk1 || pk2)` , with `pk1` and `pk2` being the public keys. This can be achieved with the following code snippet:

```c
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, crypto_generichash_BYTES);
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, crypto_generichash_BYTES);
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 */
```

Constants

- `crypto_scalarmult_BYTES`
- `crypto_scalarmult_SCALARBYTES`

Algorithm details

- X25519 (ECDH over Curve25519) - RFC 7748

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.

See 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.

In order to do so, the Blake2 hash function is an efficient alternative to the HKDF contruction:

```c
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_*`), 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:

```
#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);
```

A higher-level API will be provided in a future revision of the library in order to abstract this.

Advanced

The SHA-2 hash functions family

The SHA-256 and SHA-512 functions are provided for interoperability with other applications.

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.

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. For this purpose, use the `crypto_pwhash` API documented in the Password Hashing section.

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

Keyed message authentication using HMAC-SHA-256, HMAC-SHA-512 and HMAC-SHA512-256 (truncated HMAC-SHA-512) are provided.

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];

randombytes_buf(key, sizeof 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_hash_sha512_state state;

randombytes_buf(key, sizeof key);

crypto_hash_sha512_init(&state, key, sizeof key);

crypto_hash_sha512_update(&state, MESSAGE_PART1, MESSAGE_PART1_LEN);
crypto_hash_sha512_update(&state, MESSAGE_PART2, MESSAGE_PART2_LEN);

crypto_hash_sha512_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`.

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)`.

HMAC-SHA-512

Similarily 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);
```

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.

```
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);
```

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_hash_hmacsha*_init()` before updating or finalizing it. After `crypto_hash_hmacsha*_final()` returns, the state should not be used any more, unless it is reinitialized using `crypto_hash_hmacsha*_init()` .
- Arbitrary key lengths are supported using the multi-part interface.
- `crypto_auth_hmacsha256_*()` can be used to create AWS HmacSHA256 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.

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 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];

randombytes_buf(key, sizeof 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;

randombytes_buf(key, sizeof 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` is 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`.

`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()`.

Constants

- `crypto_onetimeauth_BYTES`
- `crypto_onetimeauth_KEYBYTES`

Data types

- `crypto_onetimeauth_state`

Algorithm details

- Poly1305

Stream ciphers

Sodium includes implementations of XSalsa20/20 and ChaCha20/20 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 that 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 uses a dedicated 64-bit block counter to avoid incrementing the nonce after each block.

Usage

```
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.

Constants

- `crypto_stream_chacha20_KEYBYTES`
- `crypto_stream_chacha20_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.

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.

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.

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);
```

- 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);
```

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.

Constants

- `crypto_stream_KEYBYTES`
- `crypto_stream_NONCEBYTES`
- `crypto_stream_PRIMITIVE`

Ed25519 to Curve25519 keys conversion

Ed25519 keys can be converted to Curve25519 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 curve25519_pk[crypto_scalarmult_curve25519_BYTES];
unsigned char curve25519_sk[crypto_scalarmult_curve25519_BYTES];

crypto_sign_ed25519_keypair(ed25519_pk, ed25519_skpk);

crypto_sign_ed25519_pk_to_curve25519(curve25519_pk, ed25519_pk);
crypto_sign_ed25519_sk_to_curve25519(curve25519_sk, ed25519_skpk);
```

Usage

```
int crypto_sign_ed25519_pk_to_curve25519(unsigned char *curve25519_pk,
                                         const unsigned char *ed25519_pk);
```

The `crypto_sign_ed25519_pk_to_curve25519()` function converts an Ed25519 public key `ed25519_pk` to a Curve25519 public key and stores it into `curve25519_pk` .

```
int crypto_sign_ed25519_sk_to_curve25519(unsigned char *curve25519_sk,
                                         const unsigned char *ed25519_sk);
```

The `crypto_sign_ed25519_sk_to_curve25519()` function converts an Ed25519 secret key `ed25519_sk` to a Curve25519 secret key and stores it into `curve25519_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.

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 no 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()` .

However, it is not thread-safe, 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. 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 compatibilility, specific implementations are intentionally not directly accessible. The library is responsible for chosing 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.

Thread safety

Initializing the random number generator is the only operation which is not thread-safe.

`sodium_init()` should be called once, before other functions. It picks the best implementations for the current platform, initializes the random number generator and generates the canary for guarded heap allocations.

After `sodium_init()` has been called, 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` .

Extra padding

For some operations, the traditional NaCl API requires extra zero bytes (`*_ZEROBYTES` , `*_BOXZEROBYTES`) before messages and ciphertexts.

However, this proved to be error-prone.

For this reason, functions whose input requires extra padding are discouraged in Sodium.

When not API compatibility is needed, alternative functions that do not require padding are available.

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.

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 to 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

Continous 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

The test suite has to always pass on the following environments:

- OpenBSD/x86_64 using `gcc -fstack-protector-strong -fstack-shuffle`
- Ubuntu/x86_64 using gcc 6, `-fsanitize=address,undefined` and Valgrind
- Ubuntu/x86_64 using clang, `-fsanitize=address,undefined` and Valgrind
- Ubuntu/x86_64 using tcc
- OSX using Xcode 7
- OSX using CompCert
- Windows 10 using Visual Studio 2010, 2012, 2013, 2015
- msys2 using mingw32 and mingw64
- ArchLinux/armv6
- Debian/sparc
- Debian/ppc
- 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.

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 curve25519 keys conversion
- iOS/Android compatibility

1.0.x roadmap

- Constant-time bin2hex() [DONE] and hex2bin() [DONE]
- Improve consistency and clarity of function prototypes
- Improve documentation
- Consider `getrandom(2)` [DONE]
- Consider Gitian
- Complete the sodium-validation project [IN PROGRESS]
- Optimized implementations for ARM w/NEON
- AVX optimized Curve25119 [DONE]
- Precomputed interface for crypto_box_easy() [DONE]
- First-class support for Javascript [DONE]
- SIMD implementations of ChaCha20 [DONE]
- SIMD implementations of Poly1305 [DONE]
- chacha20 and chacha20poly1305 with a 96 bit nonce and a 32 bit counter [DONE]
- IETF-compatible chacha20poly1305 implementation [DONE]
- Ed448-Goldilocks
- SSE-optimized BLAKE2b implementation [DONE]
- AES-GCM [DONE]
- AES-GCM detached mode [DONE]
- Montgomery reduction for GHASH

- ChaCha20-Poly1305 detached mode [DONE]
- Argon2i as crypto_pwhash [DONE]
- Multithreaded crypto_pwhash
- High-level key exchange API
- Generic subkey derivation API
- HS1-SIV or other nonce-misuse resistant scheme
- Consider SHAKE256
- BLAKE2 AVX2 implementations [DONE]
- Keyed (hash-then-encrypt) crypto_pwhash