TweetNaCl: A crypto library in 100 tweets

Daniel J. Bernstein[1,2], Wesley Janssen[3], Tanja Lange[2], and Peter Schwabe[3] ⋆

[1] Department of Computer Science
University of Illinois at Chicago, Chicago, IL 60607–7053, USA
djb@cr.yp.to
[2] Department of Mathematics and Computer Science
Technische Universiteit Eindhoven
P.O. Box 513, 5600MB Eindhoven, the Netherlands
tanja@hyperelliptic.org
[3] Radboud University Nijmegen
Digital Security Group
PO Box 9010, 6500GL Nijmegen, The Netherlands
w.janssen@student.ru.nl,peter@cryptojedi.org

Abstract. This paper introduces TweetNaCl, a compact reimplementation of the NaCl library, including all 25 of the NaCl functions used by applications. TweetNaCl is published on Twitter and fits into just 100 tweets; the tweets are available from anywhere, any time, in an unsuspicious way. Distribution via other social media, or even printed on a sheet of A4 paper, is also easily possible.

TweetNaCl is human-readable C code; it is the smallest readable implementation of a high-security cryptographic library. TweetNaCl is the first cryptographic library that allows correct functionality to be verified by human auditors with reasonable effort, making it suitable for inclusion into the trusted code base of a secure computer system.

TweetNaCl consists of a single C source file, accompanied by a single header file generated by a short Python script (1811 bytes). The library can be trivially integrated into a wide range of software build processes.

Portability and small code size come at a loss in efficiency, but TweetNaCl is sufficiently fast for most applications. TweetNaCl's cryptographic implementations meet the same security and reliability standards as NaCl: for example, complete protection against cache-timing attacks.

Keywords: trusted code base, source-code size, auditability, software implementation, timing-attack protection, NaCl, Twitter

1 Introduction

> *OpenSSL is the space shuttle of crypto libraries. It will get you to space, provided you have a team of people to push the ten thousand buttons required to do so. NaCl is more like an elevator—you just press a button and it takes you there. No frills or options.*
>
> *I like elevators.* —Matthew D. Green, 2012 [15]

Cryptographic libraries form the backbone of security applications. The Networking and Cryptography library (NaCl) [10], see nacl.cr.yp.to, is rapidly becoming the crypto library of choice for a new generation of applications. NaCl is used, for example, in BitTorrent

⋆ This work was supported by the National Science Foundation under grant 1018836 and by the Netherlands Organisation for Scientific Research (NWO) under grant 639.073.005. Permanent ID of this document: c74b5bbf605ba02ad8d9e49f04aca9a2. Date: 2013.12.29.

Live [12]; in DNSCrypt [21] from OpenDNS; in the secure mobile messaging app Threema [23]; and in the "new (faster and safer) NTor" protocol [14], the new default for Tor [24].

There are several reasons that NaCl has attracted attention. NaCl presents the developer with a high-level API: for example, all of the work necessary for signing a message is integrated into NaCl's `crypto_sign` function, and all of the work necessary for public-key authenticated encryption is integrated into NaCl's `crypto_box` function. For each of these functionalities NaCl provides exactly one default combination of cryptographic primitives selected for high security and easy protection against timing attacks. For comparison, OpenSSL [22] provides the implementor with a minefield of options, including many combinations that are broken by timing attacks and many combinations that provide no security at all.

NaCl is also much faster than OpenSSL. For example, on one core of a 2.5GHz Intel Core i5-3210M Ivy Bridge CPU, OpenSSL's RSA-2048 encryption takes 0.13 million cycles but RSA-2048 decryption takes 4.2 million cycles and elliptic-curve encryption/decryption (DH) takes 0.7 million cycles. NaCl's elliptic-curve encryption/decryption takes just 0.18 million cycles. Both NaCl and OpenSSL include optimized assembly-language implementations, but NaCl uses state-of-the-art primitives that inherently allow higher speed than the primitives included in OpenSSL: in this case, the Curve25519 elliptic curve rather than the NIST P-256 elliptic curve or lower-security RSA-2048. This performance gap is not limited to high-end Intel CPUs: see [11] for a performance analysis of the same primitives on the ARM Cortex-A8 CPU core used in the iPad 1 and iPhone 4 three years ago and in the low-cost BeagleBone Black today.

However, NaCl's performance comes at a price. A single NaCl function usually consists of several different implementations, often including multiple assembly-language implementations optimized for different CPUs. NaCl's compilation system is correspondingly complicated. Auditing the NaCl source is a time-consuming job. For example, four implementations of the `ed25519` signature system have been publicly available and waiting for integration into NaCl since 2011, but in total they consist of 5521 lines of C code and 16184 lines of `qhasm` code. Partial audits have revealed a bug in this software (`r1 += 0 + carry` should be `r2 += 0 + carry` in `amd64-64-24k`) that would not be caught by random tests; this illustrates the importance of audits. There has been some progress towards computer verification of formal proofs of correctness of software, but this progress is still far from complete verification of a usable high-security cryptographic library.

TweetNaCl: a small reimplementation of NaCl. This paper introduces TweetNaCl (pronounced "tweet salt"), a reimplementation of all 25 C NaCl functions used by applications. Each TweetNaCl function has exactly the same interface and semantics as the C NaCl function by the same name. (NaCl also includes an alpha-test networking component and support for languages other than C; TweetNaCl does not attempt to imitate these features.)

What distinguishes TweetNaCl from NaCl, and from other cryptographic libraries, is TweetNaCl's conciseness. We have posted TweetNaCl at `https://twitter.com/TweetNaCl` as a sequence of just 100 tweets. The tweets are also shown in Appendix A of this paper. The tweets, plus 1 byte at the end of each line, occupy a total of 13438 bytes.

What we actually wrote was a slightly less compact 809-line 16621-byte `tweetnacl.c`. We then wrote a simple Python script, shown in Appendix B, to remove unnecessary spaces and produce the tweet form of TweetNaCl shown in Appendix A. Developers using TweetNaCl are expected to feed the tweet form of TweetNaCl through any standard indentation program, such as the UNIX `indent` program, to produce something similar to the original `tweetnacl.c`.

An accompanying 1811-byte Python script, shown in Appendix C, prints a `tweetnacl.h` that declares all the functions in `tweetnacl.c`, together with the same set of macros provided by NaCl. NaCl actually splits these declarations and macros into a moderately large collection of `.h` files such as `crypto_box.h`, `crypto_box_curve25519xsalsa20poly1305.h`, etc.; we have a similar Python script that creates the same collection of `.h` files, but switching to `tweetnacl.h` is minimal effort for developers.

TweetNaCl is not "obfuscated C": in indented form it is easily human-readable. It does use two macros and five `typedefs`, for example to abbreviate `for (i = 0;i < n;++i)` as `FOR(i,n)` and to abbreviate `unsigned char` as `u8`, but we believe that these abbreviations improve readability, and any readers who disagree can easily remove the abbreviations.

TweetNaCl is not merely human-readable; it is human-auditable. TweetNaCl is short enough and simple enough for humans to audit against a mathematical description of the functionality in NaCl such as [2]. TweetNaCl makes it possible to audit the complete cryptographic portion of the trusted code base of a computer system. Of course, compilers also need to be audited (or to produce proofs of correct translations), as do other critical system components.

TweetNaCl is secure and reliable. TweetNaCl is a C library containing the same protections as NaCl against simple timing attacks, cache-timing attacks, etc. It has no branches depending on secret data, and it has no array indices depending on secret data. We do not want developers to be faced with a choice between TweetNaCl's conciseness and NaCl's security.

TweetNaCl is also thread-safe, and has no dynamic memory allocation. TweetNaCl, like C NaCl, stores all temporary variables in limited areas of the stack. There are no hidden failure cases: TweetNaCl reports forgeries in the same way as C NaCl, and is successful in all other cases.

TweetNaCl's functions compute the same outputs as C NaCl: the libraries are compatible. We have verified all TweetNaCl functions against the NaCl test suite.

TweetNaCl is portable and easy to integrate. Another advantage of TweetNaCl's conciseness is that developers can simply add `tweetnacl.c` and `tweetnacl.h` into their applications, without worrying about complicated configuration systems or dependencies upon external libraries. TweetNaCl works straightforwardly with a broad range of compilation systems, including cross-compilation systems, and runs on any device that can compile C. We comment that TweetNaCl also provides another form of portability, namely literal portability, while maintaining literal readability: TweetNaCl fits onto a single sheet of paper in a legible font size.

For comparison, the Sodium library from Denis [13] is a "portable, cross-compilable, installable, packageable fork of NaCl, with a compatible API"; current `libsodium-0.4.5.tar.gz` has 540467 bytes and unpacks into 381 files totaling 2524003 bytes. Many NaCl applications (e.g., DNSCrypt), and 14 NaCl bindings for various languages, are actually using Sodium. TweetNaCl is similar to Sodium in being portable, cross-compilable, installable, and packageable; but TweetNaCl has the added advantage of being so small that it can be trivially incorporated into applications by inclusion rather than by reference. We have placed TweetNaCl into the public domain, and we encourage applications to make use of it.

The first version of Sodium was obtained by reducing NaCl to its reference implementations, removing all of the optimized implementations, and simplifying the build system accordingly. We emphasize that this does not produce anything as concise as TweetNaCl. The

remaining sections of this paper describe the techniques we used to reduce the complexity of the TweetNaCl code, compared to the NaCl reference implementations.

TweetNaCl is fast enough for typical applications. TweetNaCl's focus on code size means that TweetNaCl cannot provide optimal run-time performance; NaCl's optimized assembly is often an order of magnitude faster. However, TweetNaCl is sufficiently fast for most cryptographic applications. Most applications can tolerate the 4.2 million cycles that OpenSSL uses on an Ivy Bridge CPU for RSA-2048 decryption, for example, so they can certainly tolerate the 2.5 million cycles that TweetNaCl uses for higher-security decryption (Curve25519). Note that, at a typical 2.5GHz CPU speed, this is 1000 decryptions per second per CPU core. One can of course find examples of busy applications that need the higher performance of NaCl, but those examples do not affect the usability of TweetNaCl in typical lower-volume cryptographic applications.

Of course, it would be better for compilers to turn concise source code into optimal object code, so that there is no need for optimized assembly in the first place. We leave this as a challenge for language designers and compiler writers.

TweetNaCl is also small after compilation. TweetNaCl remains reasonably small when compiled, even though this was not its primary goal. For example, when TweetNaCl is compiled with `gcc -Os` on an Intel CPU, it takes only 12745 bytes. Small compiled code has several benefits: perhaps most importantly, it avoids instruction-cache misses, both for its own startup and for other code that would otherwise have been kicked out of cache. Note that typical cryptographic benchmarks ignore these costs.

For some C compilers, putting all of TweetNaCl into a single `.c` file prevents separate linking: the final binary will include all TweetNaCl functions even if not all of those functions are used. Any developers who care about the penalty here could comment out the unused code, but TweetNaCl is so small that this penalty is negligible in the first place.

On some platforms, code is limited in total size, not just in the amount that can be cached. This was the motivation for Hutter and Schwabe to reimplement NaCl to fit into the limited flash storage and RAM available on AVR microcontrollers [18]. Their low-area implementation consists of several thousand lines written in assembly and compiles to 17366 bytes; they also have faster implementations using somewhat more area. TweetNaCl compiles to somewhat more code, 29608 bytes on the same platform, but is much easier to read and to verify, especially since the verification work for TweetNaCl is shared across platforms.

TweetNaCl is a full library, not just isolated functions. In June 2013, Green [16] announced a new contest to "identify useful cryptographic algorithms that can be formally described in one Tweet." TweetNaCl is inspired by, but not a submission to, this contest. Unlike the submissions in that Twitter thread, later submissions using `#C1T` on Twitter, or TweetCipher [1] (authenticated encryption in 6 tweets, but with an experimental cryptosystem cobbled together for the sole purpose of being short), TweetNaCl provides exactly NaCl's high-level high-security cryptographic operations. TweetNaCl includes all necessary conversions to and from wire format, modular arithmetic from scratch, etc., using nothing but the C language.

TweetNaCl provides extremely high source-code availability. In 1995, at the height of the crypto wars, the United States government regarded cryptographic software as arms and subjected it to severe export control. In response, Zimmermann published the PGP software as a printed book [27]. The export-control laws did not cover printed material, so the book

```
crypto_box = crypto_box_curve25519xsalsa20poly1305
crypto_box_open
crypto_box_keypair
crypto_box_beforenm
crypto_box_afternm
crypto_box_open_afternm
crypto_core_salsa20
crypto_core_hsalsa20
crypto_hashblocks = crypto_hashblocks_sha512
crypto_hash = crypto_hash_sha512
crypto_onetimeauth = crypto_onetimeauth_poly1305
crypto_onetimeauth_verify
crypto_scalarmult = crypto_scalarmult_curve25519
crypto_scalarmult_base
crypto_secretbox = crypto_secretbox_xsalsa20poly1305
crypto_secretbox_open
crypto_sign = crypto_sign_ed25519
crypto_sign_open
crypto_sign_keypair
crypto_stream = crypto_stream_xsalsa20
crypto_stream_xor
crypto_stream_salsa20
crypto_stream_salsa20_xor
crypto_verify_16
crypto_verify_32
```

Fig. 1. Functions supported by TweetNaCl.

could be shipped abroad. Producing usable PGP software from the printed copies (see [26]) required hours of volunteer work to OCR and proofread over 6000 pages of code.

TweetNaCl fits onto just 1 page. This conciseness opens up many new possibilities for software distribution, ensuring the permanent availability of TweetNaCl to users worldwide, even users living under regimes that have decided to censor our 100 tweets. Of course, PGP is a full-fledged cryptographic application rather than just a cryptographic library, but we expect TweetNaCl to enable a broad spectrum of *small* high-security cryptographic applications.

Functions supported by TweetNaCl. Simple NaCl applications need only six high-level NaCl functions: `crypto_box` for public-key authenticated encryption; `crypto_box_open` for verification and decryption; `crypto_box_keypair` to create a public key in the first place; and similarly `crypto_sign`, `crypto_sign_open`, `crypto_sign_keypair`.

A minimalist implementation of the NaCl API would provide just these six functions. TweetNaCl is more ambitious, supporting all 25 of the NaCl functions listed in Table 1, which as mentioned earlier are all of the C NaCl functions used by applications. This list includes all of NaCl's "default" primitives except for `crypto_auth_hmacsha512256`, which was included in NaCl only for compatibility with standards and is superseded by `crypto_onetimeauth`.

As mentioned earlier, the Ed25519 signature system has not yet been integrated into NaCl, since the Ed25519 software has not yet been fully audited; NaCl currently provides an older signature system. However, NaCl has announced that it will transition to Ed25519, so TweetNaCl provides Ed25519.

In surveying NaCl applications we have found two main reasons that applications go beyond the minimal list of six functions. First, many NaCl applications split (e.g.) `crypto_box`

into `crypto_box_beforenm` and `crypto_box_afternm` to improve speed. Second, some NaCl applications are experimenting with variations of NaCl's high-level operations but continue to use lower-level NaCl functions such as `crypto_secretbox` and `crypto_hash`.

It is important for all of these applications to continue to work with TweetNaCl. The challenge here is the code size required to provide many functions. Even a single very simple function such as

```
int crypto_box_beforenm(u8 *k,const u8 *y,const u8 *x)
{
  u8 s[32];
  crypto_scalarmult(s,x,y);
  return crypto_core_hsalsa20(k,z,s,sigma);
}
```

costs us approximately 1 tweet. We could use shorter function names internally, but we would then need further wrappers to provide all the external function names listed in Table 1. We have many such functions, and a limited tweet budget, limiting the space available for actual cryptographic computations.

2 Salsa20, HSalsa20, and XSalsa20

NaCl encrypts messages by xor'ing them with the output of Bernstein's Salsa20 [5] stream cipher. The Salsa20 stream cipher generates 64-byte output blocks using the Salsa20 "core function" in counter mode. The main loop in NaCl's reference implementation of the Salsa20 core function, `crypto_core/salsa20/ref/core.c`, transforms 16 32-bit words `x0`, `x1`, ..., `x15` as follows, where `ROUNDS` is 20:

```
for (i = ROUNDS;i > 0;i -= 2) {
   x4 ^= rotate( x0+x12, 7);  x8 ^= rotate( x4+ x0, 9);
  x12 ^= rotate( x8+ x4,13);  x0 ^= rotate(x12+ x8,18);
   x9 ^= rotate( x5+ x1, 7); x13 ^= rotate( x9+ x5, 9);
   x1 ^= rotate(x13+ x9,13);  x5 ^= rotate( x1+x13,18);
  x14 ^= rotate(x10+ x6, 7);  x2 ^= rotate(x14+x10, 9);
   x6 ^= rotate( x2+x14,13); x10 ^= rotate( x6+ x2,18);
   x3 ^= rotate(x15+x11, 7);  x7 ^= rotate( x3+x15, 9);
  x11 ^= rotate( x7+ x3,13); x15 ^= rotate(x11+ x7,18);
   x1 ^= rotate( x0+ x3, 7);  x2 ^= rotate( x1+ x0, 9);
   x3 ^= rotate( x2+ x1,13);  x0 ^= rotate( x3+ x2,18);
   x6 ^= rotate( x5+ x4, 7);  x7 ^= rotate( x6+ x5, 9);
   x4 ^= rotate( x7+ x6,13);  x5 ^= rotate( x4+ x7,18);
  x11 ^= rotate(x10+ x9, 7);  x8 ^= rotate(x11+x10, 9);
   x9 ^= rotate( x8+x11,13); x10 ^= rotate( x9+ x8,18);
  x12 ^= rotate(x15+x14, 7); x13 ^= rotate(x12+x15, 9);
  x14 ^= rotate(x13+x12,13); x15 ^= rotate(x14+x13,18);
}
```

Notice that this loop involves 96 `x` indices: `x4`, `x0`, `x12`, `x8`, `x4`, etc. TweetNaCl handles the same loop much more concisely:

```
FOR(i,20) {
  FOR(j,4) {
    FOR(m,4) t[m] = x[(5*j+4*m)%16];
    t[1] ^= rotate(t[0]+t[3], 7); t[2] ^= rotate(t[1]+t[0], 9);
    t[3] ^= rotate(t[2]+t[1],13); t[0] ^= rotate(t[3]+t[2],18);
    FOR(m,4) w[4*j+(j+m)%4] = t[m];
  }
  FOR(m,16) x[m] = w[m];
}
```

We emphasize two levels of Salsa20 symmetry that appear in the Salsa20 specification and that are expressed explicitly in this TweetNaCl loop. First, the 20 rounds in Salsa20 alternate between "column rounds" and "row rounds", with column rounds operating on columns of the 4×4 matrix

$$\begin{pmatrix} \texttt{x[0]} & \texttt{x[1]} & \texttt{x[2]} & \texttt{x[3]} \\ \texttt{x[4]} & \texttt{x[5]} & \texttt{x[6]} & \texttt{x[7]} \\ \texttt{x[8]} & \texttt{x[9]} & \texttt{x[10]} & \texttt{x[11]} \\ \texttt{x[12]} & \texttt{x[13]} & \texttt{x[14]} & \texttt{x[15]} \end{pmatrix}$$

and row rounds operating in exactly the same way on rows of the matrix. TweetNaCl computes a row round as a transposition of the matrix followed by a column round followed by another transposition; i.e., the 20 rounds consist of 20 iterations of "compute a column round and transpose the output". The transposed result of each round is built in a separate array `w` to avoid overwriting the round input; it is then copied from `w` back to `x`. One can easily see that the indices `4*j+(j+m)%4` for `w` are the transposes of the indices `(5*j+4*m)%16` for `x`.

Second, the column round operates on the column down from `x[0]`, operates in the same way on the column down from `x[5]` (wrapping around to `x[1]`), operates in the same way on the column down from `x[10]`, and operates in the same way on the column down from `x[15]`. TweetNaCl has `j` loop over the 4 columns; the `x` index `(5*j+4*m)%16` is `m` columns down from the starting point in column `j`.

For comparison, the indices in the second half of the NaCl loop shown above are the transposes of the indices in the first half, and the indices in the first half have these symmetries across columns. Verifying these 96 indices is of course feasible but takes considerably more time than verifying the corresponding segment of TweetNaCl code—and this is just the first of many ways in which NaCl's reference implementations consume more code than TweetNaCl.

Stream generation and stream encryption. NaCl actually has two ways to use Salsa20: `crypto_stream_salsa20` produces any desired number of bytes of the Salsa20 output stream; `crypto_stream_salsa20_xor` produces a ciphertext from a plaintext. Both of these functions are wrappers around `crypto_core_salsa20`; both functions handle initialization and updates of the block counter, and output lengths that are not necessarily multiples of 64. The difference is that the second function xors each block with a plaintext block, moving along the plaintext accordingly.

In TweetNaCl, `crypto_stream_salsa20` simply calls `crypto_stream_salsa20_xor` with a null pointer for the plaintext. This eliminates essentially all the duplication of code between these two functions, at the expense of three small tweaks to `crypto_stream_salsa20_xor`, such as replacing

```
FOR(i,64) c[i] = m[i] ^ x[i];
```

with

```
FOR(i,64) c[i] = (m?m[i]:0) ^ x[i];
```

to treat a null pointer `m` as if it were a pointer to an all-0 block.

XSalsa20 and HSalsa20. NaCl's `crypto_stream` actually uses Bernstein's XSalsa20 stream cipher (see [6]) rather than the Salsa20 stream cipher. The difference is that XSalsa20 supports 32 bytes of nonce/counter input while Salsa20 supports only 16 bytes of nonce/counter input. XSalsa20 uses the original 32-byte key and the first 16 bytes of the nonce to generate an intermediate 32-byte key, and then uses Salsa20 with the intermediate key and the remaining 16 bytes of nonce/counter to generate each output block.

The intermediate key generation, called "HSalsa20", is similar to Salsa20 but slightly more efficient, and has a separate implementation in NaCl. For our purposes this is a problem: it means almost doubling the code size.

TweetNaCl does better by viewing HSalsa20 as (1) generating a 64-byte Salsa20 output block, (2) extracting 32 bytes from particular output positions, and then (3) transforming those 32 bytes in a public invertible way. The transformation is much more concise than a separate HSalsa20 implementation, allowing TweetNaCl to implement both `crypto_core_salsa20` and `crypto_core_hsalsa20` as wrappers around a unified `core` function.

We do not claim novelty for this view of HSalsa20: the same structure is exactly what allowed the proof in [6] that the security of Salsa20 implies the security of HSalsa20 and XSalsa20. What is new is the use of this structure to simplify a unified Salsa20/HSalsa20 implementation.

3 Poly1305

Secret-key authentication in NaCl uses Bernstein's Poly1305 [3] authenticator. The Poly1305 code in the NaCl reference implementation is already quite concise. For elements of $\mathbb{F}_{2^{130}-5}$ it uses a radix-2^8 representation; we use the same representation for TweetNaCl.

The NaCl reference implementation uses a `mulmod` function for multiplication in $\mathbb{F}_{2^{130}-5}$, a `squeeze` function to perform two carry chains after multiplication and a `freeze` function to produce a unique representation of an element of $\mathbb{F}_{2^{130}-5}$. Each of these functions is called only once in the Poly1305 main loop; we inline those functions to remove code for the function header and the call. The reference implementation also uses an `add` function which is called once in the main loop, once during finalization and once inside the `freeze` function. We keep the function, but rename it to `add1305` to avoid confusion with the `add` function used (as described in Section 5) for elliptic-curve addition.

We furthermore shorten the code of modular multiplication. NaCl's reference implementation performs multiplication of `h` by `r` with the result in `hr` as follows:

```
for (i = 0;i < 17;++i) {
  u = 0;
  for (j = 0;j <= i;++j)
    u += h[j] * r[i - j];
  for (j = i + 1;j < 17;++j)
    u += 320 * h[j] * r[i + 17 - j];
  hr[i] = u;
}
```

This piece of code exploits the fact that $2^{136} \equiv 320 \pmod{2^{130}-5}$ for modular reduction on the fly. TweetNaCl merges the two inner loops:

```
FOR (i, 17) {
  x[i] = 0;
  FOR (j, 17)
    x[i] += h[j] * ((j <= i) ? r[i - j] : 320 * r[i + 17 - j]);
}
```

4 SHA-512

The default hash function in NaCl and the hash function used within the Ed25519 signature scheme (see Section 5) is SHA-512 [25]. The SHA-512 code in the NaCl reference implementation consists of two main portions of code:

- The function `crypto_hash`, which performs initialization of the hash value with the IV and computation of padding; and
- the `crypto_hashblocks` function which performs hashing of full blocks.

Padding. Outside of `crypto_hashblocks`, the most complex part of `crypto_hash` is the message padding. The reference padding code, with TweetNaCl's choices of variable names substituted for the original choices, is as follows:

```
for (i = 0;i < n;++i) x[i] = m[i];
x[n] = 0x80;
if (n < 112) {
  for (i = n + 1;i < 119;++i) x[i] = 0;
  x[119] = b >> 61;
  x[120] = b >> 53; x[121] = b >> 45;
  x[122] = b >> 37; x[123] = b >> 29;
  x[124] = b >> 21; x[125] = b >> 13;
  x[126] = b >> 5;  x[127] = b << 3;
  crypto_hashblocks(h,x,128);
} else {
  for (i = n + 1;i < 247;++i) x[i] = 0;
  x[247] = b >> 61;
  x[248] = b >> 53; x[249] = b >> 45;
  x[250] = b >> 37; x[251] = b >> 29;
  x[252] = b >> 21; x[253] = b >> 13;
  x[254] = b >> 5;  x[255] = b << 3;
  crypto_hashblocks(h,x,256);
}
```

This segment handles two possibilities for processing the final partial block of SHA-512 input: if the block has fewer than 112 bytes then it is padded to 128 bytes; otherwise it is padded to 256 bytes. The padding ends with a 9-byte big-endian encoding of the number of message bits.

TweetNaCl simplifies this code in three ways. First, it eliminates the two separate lines of zero-padding `x` in favor of initializing the whole array to 0. Second, elsewhere in TweetNaCl there is a `ts64` function (used at the end of the SHA-512 compression function) that stores 64 bits in big-endian form; TweetNaCl reuses this function inside the padding. Third, TweetNaCl merges the two branches, reusing `n` (which has no later use) for the number of bytes in the padded block. The final padding code is much more concise than the original:

```
FOR(i,256) x[i] = 0;
FOR(i,n) x[i] = m[i];
x[n] = 128;
n = 256-128*(n<112);
x[n-9] = b >> 61;
ts64(x+n-8,b << 3);
crypto_hashblocks(h,x,n);
```

Hashing blocks. SHA-512 performs 80 rounds of computation per block. The NaCl reference implementation has 80 lines for these 80 rounds. Each round is just one invocation of an `F` macro (interruped by invocations of an `EXPAND` macro after every 16 rounds), but this still results in a significant amount of code. TweetNaCl instead uses a loop over the 80 rounds. With such a "rolled" loop there is only one invocation of each of the macros, so TweetNaCl inlines those.

In NaCl the 16 64-bit message words are loaded into variables `w0`, `w1`, ..., `w15`; the internal temporary state is kept in variables `a`, `b`, ..., `h`. TweetNaCl uses arrays `u64 w[16]` and `u64 a[8]` instead. This allows us to also roll all initialization and copy loops. The final code for processing one 128-byte block is the following:

```
FOR(i,16) w[i] = dl64(m + 8 * i);

FOR(i,80) {
  FOR(j,8) b[j] = a[j];
  t = a[7] + Sigma1(a[4]) + Ch(a[4],a[5],a[6]) + K[i] + w[i%16];
  b[7] = t + Sigma0(a[0]) + Maj(a[0],a[1],a[2]);
  b[3] += t;
  FOR(j,8) a[(j+1)%8] = b[j];
  if (i%16 == 15)
    FOR(j,16)
      w[j] += w[(j+9)%16] + sigma0(w[(j+1)%16]) + sigma1(w[(j+14)%16]);
}

FOR(i,8) { a[i] += z[i]; z[i] = a[i]; }
```

Obviously there is still some complexity in this code, but this directly reflects the inherent complexity of the SHA-512 function; the SHA-512 specification [25] is easily verified to match TweetNaCl's implementation. The functions `Sigma1`, `Ch`, `Sigma0`, `Maj`, `sigma0`, and `sigma1` are one-line implementations of the functions Σ_1, Ch, Σ_0, Maj, σ_0, and σ_1 from the SHA-512 specification.

5 Curve25519 and Ed25519

Asymmetric cryptography in NaCl uses Bernstein's Curve25519 elliptic-curve Diffie-Hellman key exchange [4] and will use the Ed25519 elliptic-curve signature scheme from Bernstein, Duif, Lange, Schwabe, and Yang [7,8]. This section explains the techniques we use for our compact implementation of these two schemes.

Arithmetic in $\mathbb{F}_{2^{255}-19}$. Both Curve25519 and Ed25519 require arithmetic in the field $\mathbb{F}_{2^{255}-19}$. We represent an element of this finite field as an array of 16 signed 64-bit integers (datatype `signed long long`) in radix 2^{16}:

```
typedef i64 gf[16];
```

Additions and subtractions do not have to worry about carries or modular reduction; they simply turn into a loop that performs 16 coefficient additions or subtractions.

Multiplication performs simple "operand scanning" schoolbook multiplication in two nested loops. We then reduce modulo $2^{256} - 38$:

```
i64 i,j,t[31];
FOR(i,31) t[i]=0;
FOR(i,16) FOR(j,16) t[i+j]+=a[i]*b[j];
FOR(i,15) t[i]+=38*t[i+16];
FOR(i,16) o[i]=t[i];
```

The 16 result coefficients in `o` are too large to be used as input to another multiplication. We use two calls to a `car25519` carry function to solve this problem. This carry function modifies the result `o` in place as follows:

```
FOR(i,16) {
  o[i] += (1LL<<16);
  c = o[i]>>16;
  o[(i+1)*(i<15)] += c-1+37*(c-1)*(i==15);
  o[i] -= c<<16;
}
```

Aside from carrying from one limb to the next, the function also adds 2^{16} to each limb and subtracts 1 from the next highest limb before performing the carry. This ensures that repeated application of the function brings all limbs into the interval $[0, 2^{16}-1]$. Without this addition, repeated application of the carry chain would bring all limbs into the interval $[-2^{16}-1, 2^{16}-1]$. We use this additional functionality to "freeze" field elements to a unique representation at the very end of the Curve25519 or Ed25519 computations.

We reuse the multiplication for squarings, but make squarings explicit by spending a few bytes of source code for a separate function that simply calls multiplication. This makes it easy during code audit to compare the code to elliptic-curve addition formulas, for example from the Explicit Formulas Database [9]. To match the notation of [9] we use the names `M` and `S` for functions that multiply and square in the field $\mathbb{F}_{2^{255}-19}$; we also use `A` for addition and `Z` for subtraction.

Inversion uses Fermat's little theorem and is implemented through exponentation with $2^{255} - 21$. We use a simple square-and-multiply algorithm and avoid storing the exponent by making use of its special shape: it has all bits set but the bits at position 4 and position 2. We perform the square-and-multiply loop for inversion as follows:

```
for(a=253;a>=0;a--) {
  S(c,c);
  if(a!=2&&a!=4) M(c,c,i);
}
```

The square-root computation for elliptic-curve point decompression in Ed25519 uses exponentiation by $2^{252} - 3$. See [7, Section 5]. Observe that this exponent has all bits set but the bit at position 1; we use the same approach as for inversion.

Curve arithmetic. The typical Curve25519 implementation computes a Montgomery ladder [19] on the Montgomery curve $M : y^2 = x^3 + 486662x^2 + x$. The Ed25519 signature scheme performs arithmetic on the birationally equivalent twisted Edwards curve $E : -x^2 + y^2 = 1 - \frac{121665}{121666}x^2y^2$. More specifically, Ed25519 key generation and signing perform a fixed-basepoint scalar multiplication; verification performs a double-scalar multiplication.

In principle we could use the same scalar-multiplication code for both Curve25519 and Ed25519. This would require conversion of points on M to points on E and back. If we used the x-coordinate-based differential addition ladder of Curve25519 also for Ed25519, we would additionally need code to recover the y-coordinate as described by Okeya and Sakurai in [20]. Conversion code is not substantially shorter than the code required for the Curve25519 Montgomery ladder, so we decided to *not* use the same code for scalar multiplication in Curve25519 and Ed25519.

Curve25519 uses the same Montgomery ladder as the reference implementation, except that we do not use a dedicated function for multiplication by the constant 121666. For Ed25519 we decided to use only one scalar-multiplication routine that can be used in key generation, signing, and verification. We represent points on E in extended coordinates as described in [17] and implement the complete addition law in an `add` function. We use this function for both addition and doubling of points. The scalar multiplication then performs a ladder of 256 steps; each step performs an addition and a doubling:

```
set25519(p[0],gf0);
set25519(p[1],gf1);
set25519(p[2],gf1);
set25519(p[3],gf0);
for (i = 255;i >= 0;--i) {
  u8 b = (s[i/8]>>(i&7))&1;
  cswap(p,q,b);
  add(q,p);
  add(p,p);
  cswap(p,q,b);
}
```

The first four lines set the point `p` to the neutral element. The `cswap` function performs a constant-time conditional swap of `p` and `q` depending on the scalar bit that has been extracted into `b` before. The constant-time swap calls `sel25519` for each of the 4 coordinates of `p` and `q`. The function `sel25519` is reused in conditional swaps for the Montgomery ladder in Curve25519 and performs a constant-time conditional swap of field elements as follows:

```
sv sel25519(gf p,gf q,int b)
```

```
{
  i64 t,i,c=~(b-1);
  FOR(i,16) {
    t = c & (p[i]^q[i]);
    p[i] ^= t;
    q[i] ^= t;
  }
}
```

Arithmetic modulo the group order. Signing requires reduction of a 512-bit integer modulo the order of the Curve25519 group, a prime $p = 2^{252} + \delta$ where $\delta \approx 2^{124.38}$. We store this integer as a sequence of limbs in radix 2^8. We eliminate the top limb of the integer, say $2^{504}b$, by subtracting $2^{504}b$ and also subtracting $2^{252}\delta b$; we then perform a partial carry so that 20 consecutive limbs are each between -2^7 and 2^7. We repeat this procedure to eliminate subsequent limbs from the top. This is considerably more concise than typical reduction methods:

```
for (i = 63;i >= 32;--i) {
  carry = 0;
  for (j = i - 32;j < i - 12;++j) {
    x[j] += carry - 16 * x[i] * L[j - (i - 32)];
    carry = (x[j] + 128) >> 8;
    x[j] -= carry << 8;
  }
  x[j] += carry;
  x[i] = 0;
}
```

We similarly eliminate any remaining multiple of 2^{252}, leaving an integer between $-1.1 \cdot 2^{251}$ and $1.1 \cdot 2^{251}$. We then multiply the final carry bit by p and add, obtaining an integer between 0 and $p - 1$, and carry in the traditional way so that each limb is between 0 and 255.

References

1. Jean-Philippe Aumasson. Tweetcipher! (crypto challenge), 2013. http://cybermashup.com/2013/06/12/tweetcipher-crypto-challenge/ (accessed 2013-11-20). 4
2. Daniel J. Bernstein. Cryptography in NaCl. http://cr.yp.to/highspeed/naclcrypto-20090310.pdf (accessed 2013-11-20). 3
3. Daniel J. Bernstein. The Poly1305-AES message-authentication code. In Henri Gilbert and Helena Handschuh, editors, *Fast Software Encryption*, volume 3557 of *LNCS*, pages 32–49. Springer, 2005. http://cr.yp.to/papers.html#poly1305. 8
4. Daniel J. Bernstein. Curve25519: new Diffie-Hellman speed records. In Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin, editors, *Public Key Cryptography – PKC 2006*, volume 3958 of *LNCS*, pages 207–228. Springer, 2006. http://cr.yp.to/papers.html#curve25519. 11
5. Daniel J. Bernstein. The Salsa20 family of stream ciphers. In Matthew Robshaw and Olivier Billet, editors, *New stream cipher designs*, volume 4986 of *LNCS*, pages 84–97. Springer, 2008. http://cr.yp.to/papers.html#salsafamily. 6
6. Daniel J. Bernstein. Extending the Salsa20 nonce. In *Workshop record of Symmetric Key Encryption Workshop 2011*, 2011. http://cr.yp.to/papers.html#xsalsa. 8
7. Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. In Bart Preneel and Tsuyoshi Takagi, editors, *Cryptographic Hardware and Embedded Systems – CHES 2011*, volume 6917 of *LNCS*, pages 124–142. Springer, 2011. see also full version [8]. 11, 12, 14

8. Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. *Journal of Cryptographic Engineering*, 2(2):77–89, 2012. http://cryptojedi.org/papers/#ed25519, see also short version [7]. 11, 13
9. Daniel J. Bernstein and Tanja Lange. Explicit-formulas database. http://www.hyperelliptic.org/EFD/ (accessed 2013-11-20). 11
10. Daniel J. Bernstein, Tanja Lange, and Peter Schwabe. The security impact of a new cryptographic library. In Alejandro Hevia and Gregory Neven, editors, *Progress in Cryptology – LATINCRYPT 2012*, volume 7533 of *LNCS*, pages 159–176. Springer, 2012. http://cryptojedi.org/papers/#coolnacl. 1
11. Daniel J. Bernstein and Peter Schwabe. NEON crypto. In Emmanuel Prouff and Patrick Schaumont, editors, *Cryptographic Hardware and Embedded Systems – CHES 2012*, volume 7428 of *LNCS*, pages 320–339. Springer, 2012. http://cryptojedi.org/papers/#neoncrypto. 2
12. BitTorrent Live. http://live.bittorrent.com/ (accessed 2013-11-20). 2
13. Frank Denis. Introducing Sodium, a new cryptographic library, 2013. http://labs.umbrella.com/2013/03/06/announcing-sodium-a-new-cryptographic-library/ (accessed 2013-11-20). 3
14. Roger Dingledine. Tor 0.2.4.17-rc is out. Posting in [tor-talk], 2013. https://lists.torproject.org/pipermail/tor-talk/2013-September/029857.html. 2
15. Matthew Green. The anatomy of a bad idea, 2012. http://blog.cryptographyengineering.com/2012/12/the-anatomy-of-bad-idea.html (accessed 2013-11-20). 1
16. Matthew Green. Announcing a contest: identify useful cryptographic algorithms that can be formally described in one Tweet, 2013. https://twitter.com/matthew_d_green/status/342755869110464512 (accessed 2013-11-20). 4
17. Huseyin Hisil, Kenneth Koon-Ho Wong, Gary Carter, , and Ed Dawson. Twisted Edwards curves revisited. In Josef Pieprzyk, editor, *Advances in Cryptology – ASIACRYPT 2008*, volume 5350 of *LNCS*, pages 326–343. Springer, 2008. http://eprint.iacr.org/2008/522/. 12
18. Michael Hutter and Peter Schwabe. NaCl on 8-bit AVR Microcontrollers. In Amr Youssef and Abderrahmane Nitaj, editors, *Progress in Cryptology – AFRICACRYPT 2013*, volume 7918 of *LNCS*, pages 156–172. Springer, 2013. http://cryptojedi.org/papers/#avrnacl. 4
19. Peter L. Montgomery. Speeding the Pollard and elliptic curve methods of factorization. *Mathematics of Computation*, 48(177):243–264, 1987. http://www.ams.org/journals/mcom/1987-48-177/S0025-5718-1987-0866113-7/S0025-5718-1987-0866113-7.pdf. 12
20. Katsuyuki Okeya and Kouichi Sakurai. Efficient elliptic curve cryptosystems from a scalar multiplication algorithm with recovery of the y-coordinate on a Montgomery-form elliptic curve. In Çetin K. Koç, David Naccache, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems – CHES 2001*, volume 2162 of *LNCS*, pages 126–141. Springer, 2001. 12
21. Introducing DNSCrypt (preview release). http://www.opendns.com/technology/dnscrypt/ (accessed 2013-11-20). 2
22. OpenSSL. OpenSSL: The open source toolkit for SSL/TLS. http://www.openssl.org/ (accessed 2013-11-20). 2
23. Threema – seriously secure mobile messaging. https://threema.ch/en/ (accessed 2013-11-20). 2
24. Tor project: Anonymity online. https://www.torproject.org/ (accessed 2013-11-20). 2
25. U.S. DEPARTMENT OF COMMERCE/National Institute of Standards and Technology. *Secure Hash Standard (SHS)*, 2012. Federal Information Processing Standards Publication 180-4, http://csrc.nist.gov/publications/fips/fips180-4/fips-180-4.pdf. 9, 10
26. Ståle Schumacher Ytteborg. The PGPi scanning project. http://www.pgpi.org/pgpi/project/scanning/ (accessed 2013-11-20). 5
27. Philip Zimmermann. *PGP Source Code and Internals*. MIT Press, 1995. 4

A The 100 tweets

```
#include "tweetnacl.h"
#define FOR(i,n) for (i = 0;i < n;++i)
#define sv static void
typedef unsigned char u8;typedef unsigned long u32;typedef unsigned long long u64;typedef long long i64;typedef i64 gf[16];extern void
randombytes(u8*,u64);static const u8 _0[16],_9[32]={9};static const gf gf0,gf1={1},_121665={0xDB41,1},D={0x78a3,0x1359,0x4dca,0x75eb,0xd8ab,
0x4141,0x0a4d,0x0070,0xe898,0x7779,0x4079,0x8cc7,0xfe73,0x2b6f,0x6cee,0x5203},D2={0xf159,0x26b2,0x9b94,0xebd6,0xb156,0x8283,0x149a,0x00e0,
0xd130,0xeef3,0x80f2,0x198e,0xfce7,0x56df,0xd9dc,0x2406},X={0xd51a,0x8f25,0x2d60,0xc956,0xa7b2,0x9525,0xc760,0x692c,0xdc5c,0xfdd6,0xe231,
0xc0a4,0x53fe,0xcd6e,0x36d3,0x2169},Y={0x6658,0x6666,0x6666,0x6666,0x6666,0x6666,0x6666,0x6666,0x6666,0x6666,0x6666,0x6666,0x6666,0x6666,
0x6666,0x6666},I={0xa0b0,0x4a0e,0x1b27,0xc4ee,0xe478,0xad2f,0x1806,0x2f43,0xd7a7,0x3dfb,0x0099,0x2b4d,0xdf0b,0x4fc1,0x2480,0x2b83};static
u32 L32(u32 x,int c){return(x<<c)|(x>>(32-c));}static u32 ld32(const u8*x){u32 u=x[3];u=(u<<8)|x[2];u=(u<<8)|x[1];return(u<<8)|x[0];}static
u64 dl64(const u8*x){u64 i,u=0;FOR(i,8)u=(u<<8)|x[i];return u;}sv st32(u8*x,u32 u){int i;FOR(i,4){x[i]=u;u>>=8;}}sv ts64(u8*x,u64 u){int i;
for(i=7;i>=0;--i){x[i]=u;u>>=8;}}static int vn(const u8*x,const u8*y,int n){u32 i,d=0;FOR(i,n)d|=x[i]^y[i];return(1&((d-1)>>8))-1;}int
crypto_verify_16(const u8*x,const u8*y){return vn(x,y,16);}int crypto_verify_32(const u8*x,const u8*y){return vn(x,y,32);}sv core(u8*out,
const u8*in,const u8*k,const u8*c,int h){u32 w[16],x[16],y[16],t[4];int i,j,m;FOR(i,4){x[5*i]=ld32(c+4*i);x[1+i]=ld32(k+4*i);x[6+i]=ld32(in+
4*i);x[11+i]=ld32(k+16+4*i);}FOR(i,16)y[i]=x[i];FOR(i,20){FOR(j,4){FOR(m,4)t[m]=x[(5*j+4*m)%16];t[1]^=L32(t[0]+t[3],7);t[2]^=L32(t[1]+t[0],9
);t[3]^=L32(t[2]+t[1],13);t[0]^=L32(t[3]+t[2],18);}FOR(m,4)w[4*j+(j+m)%4]=t[m];}FOR(m,16)x[m]=w[m];}if(h){FOR(i,16)x[i]+=y[i];FOR(i,4){x[5*i]
-=ld32(c+4*i);x[6+i]-=ld32(in+4*i);}FOR(i,4){st32(out+4*i,x[5*i]);st32(out+16+4*i,x[6+i]);}}else FOR(i,16)st32(out+4*i,x[i]+y[i]);}int
crypto_core_salsa20(u8*out,const u8*in,const u8*k,const u8*c){core(out,in,k,c,0);return 0;}int crypto_core_hsalsa20(u8*out,const u8*in,const
u8*k,const u8*c){core(out,in,k,c,1);return 0;}static const u8 sigma[16]="expand 32-byte k";int crypto_stream_salsa20_xor(u8*c,const u8*m,u64
b,const u8*n,const u8*k){u8 z[16],x[64];u32 u,i;if(!b)return 0;FOR(i,16)z[i]=0;FOR(i,8)z[i]=n[i];while(b>=64){crypto_core_salsa20(x,z,k,
sigma);FOR(i,64)c[i]=(m?m[i]:0)^x[i];u=1;for(i=8;i<16;++i){u+=(u32)z[i];z[i]=u;u>>=8;}b-=64;c+=64;if(m)m+=64;}if(b){crypto_core_salsa20(x,z,
k,sigma);FOR(i,b)c[i]=(m?m[i]:0)^x[i];}return 0;}int crypto_stream_salsa20(u8*c,u64 d,const u8*n,const u8*k){return
crypto_stream_salsa20_xor(c,0,d,n,k);}int crypto_stream(u8*c,u64 d,const u8*n,const u8*k){u8 s[32];crypto_core_hsalsa20(s,n,k,sigma);return
crypto_stream_salsa20(c,d,n+16,s);}int crypto_stream_xor(u8*c,const u8*m,u64 d,const u8*n,const u8*k){u8 s[32];crypto_core_hsalsa20(s,n,k,
sigma);return crypto_stream_salsa20_xor(c,m,d,n+16,s);}sv add1305(u32*h,const u32*c){u32 j,u=0;FOR(j,17){u+=h[j]+c[j];h[j]=u&255;u>>=8;}}
static const u32 minusp[17]={5,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,252};int crypto_onetimeauth(u8*out,const u8*m,u64 n,const u8*k){u32 s,i,j,u,x[
17],r[17],h[17],c[17],g[17];FOR(j,17)r[j]=h[j]=0;FOR(j,16)r[j]=k[j];r[3]&=15;r[4]&=252;r[7]&=15;r[8]&=252;r[11]&=15;r[12]&=252;r[15]&=15;
while(n>0){FOR(j,17)c[j]=0;for(j=0;(j<16)&&(j<n);++j)c[j]=m[j];c[j]=1;m+=j;n-=j;add1305(h,c);FOR(i,17){x[i]=0;FOR(j,17)x[i]+=h[j]*((j<=i)?r[
i-j]:320*r[i+17-j]);}FOR(i,17)h[i]=x[i];u=0;FOR(j,16){u+=h[j];h[j]=u&255;u>>=8;}u+=h[16];h[16]=u&3;u=5*(u>>2);FOR(j,16){u+=h[j];h[j]=u&255;u
>>=8;}u+=h[16];h[16]=u;}FOR(j,17)g[j]=h[j];add1305(h,minusp);s= -(h[16]>>7);FOR(j,17)h[j]^=s&(g[j]^h[j]);FOR(j,16)c[j]=k[j+16];c[16]=0;
add1305(h,c);FOR(j,16)out[j]=h[j];return 0;}int crypto_onetimeauth_verify(const u8*h,const u8*m,u64 n,const u8*k){u8 x[16];
crypto_onetimeauth(x,m,n,k);return crypto_verify_16(h,x);}int crypto_secretbox(u8*c,const u8*m,u64 d,const u8*n,const u8*k){int i;if(d<32)
return-1;crypto_stream_xor(c,m,d,n,k);crypto_onetimeauth(c+16,c+32,d-32,c);FOR(i,16)c[i]=0;return 0;}int crypto_secretbox_open(u8*m,const u8
*c,u64 d,const u8*n,const u8*k){int i;u8 x[32];if(d<32)return-1;crypto_stream(x,32,n,k);if(crypto_onetimeauth_verify(c+16,c+32,d-32,x)!=0)
return-1;crypto_stream_xor(m,c,d,n,k);FOR(i,32)m[i]=0;return 0;}sv set25519(gf r,const gf a){int i;FOR(i,16)r[i]=a[i];}sv car25519(gf o){int
i;i64 c;FOR(i,16){o[i]+=(1LL<<16);c=o[i]>>16;o[(i+1)*(i<15)]+=c-1+37*(c-1)*(i==15);o[i]-=c<<16;}}sv sel25519(gf p,gf q,int b){i64 t,i,c=~(b-
1);FOR(i,16){t=c&(p[i]^q[i]);p[i]^=t;q[i]^=t;}}sv pack25519(u8*o,const gf n){int i,j,b;gf m,t;FOR(i,16)t[i]=n[i];car25519(t);car25519(t);
car25519(t);FOR(j,2){m[0]=t[0]-0xffed;for(i=1;i<15;i++){m[i]=t[i]-0xffff-((m[i-1]>>16)&1);m[i-1]&=0xffff;}m[15]=t[15]-0x7fff-((m[14]>>16)&1)
;b=(m[15]>>16)&1;m[15]&=0xffff;sel25519(t,m,1-b);}FOR(i,16){o[2*i]=t[i]&0xff;o[2*i+1]=t[i]>>8;}}static int neq25519(const gf a,const gf b){
u8 c[32],d[32];pack25519(c,a);pack25519(d,b);return crypto_verify_32(c,d);}static u8 par25519(const gf a){u8 d[32];pack25519(d,a);return d[0
]&1;}sv unpack25519(gf o,const u8*n){int i;FOR(i,16)o[i]=n[2*i]+((i64)n[2*i+1]<<8);o[15]&=0x7fff;}sv A(gf o,const gf a,const gf b){int i;FOR
(i,16)o[i]=a[i]+b[i];}sv Z(gf o,const gf a,const gf b){int i;FOR(i,16)o[i]=a[i]-b[i];}sv M(gf o,const gf a,const gf b){i64 i,j,t[31];FOR(i,
31)t[i]=0;FOR(i,16)FOR(j,16)t[i+j]+=a[i]*b[j];FOR(i,15)t[i]+=38*t[i+16];FOR(i,16)o[i]=t[i];car25519(o);car25519(o);}sv S(gf o,const gf a){M(
o,a,a);}sv inv25519(gf o,const gf i){gf c;int a;FOR(a,16)c[a]=i[a];for(a=253;a>=0;a--){S(c,c);if(a!=2&&a!=4)M(c,c,i);}FOR(a,16)o[a]=c[a];}sv
pow2523(gf o,const gf i){gf c;int a;FOR(a,16)c[a]=i[a];for(a=250;a>=0;a--){S(c,c);if(a!=1)M(c,c,i);}FOR(a,16)o[a]=c[a];}int
crypto_scalarmult(u8*q,const u8*n,const u8*p){u8 z[32];i64 x[96],r,i;gf a,b,c,d,e,f;FOR(i,31)z[i]=n[i];z[31]=(n[31]&127)|64;z[0]&=248;
unpack25519(x,p);FOR(i,16){b[i]=x[i];d[i]=a[i]=c[i]=0;}a[0]=d[0]=1;for(i=254;i>=0;--i){r=(z[i>>3]>>(i&7))&1;sel25519(a,b,r);sel25519(c,d,r);
A(e,a,c);Z(a,a,c);A(c,b,d);Z(b,b,d);S(d,e);S(f,a);M(c,a,e);M(a,a,c);Z(a,a,c);S(b,a);Z(c,d,f);M(a,c,_121665);A(a,a,d);M(c,c,a);M(a,d
,f);M(d,b,x);S(b,e);sel25519(a,b,r);sel25519(c,d,r);}FOR(i,16){x[i+32]=a[i];x[i+48]=c[i];x[i+64]=b[i];x[i+80]=d[i];}inv25519(x+48,x+48);M(x+
32,x+32,x+48);pack25519(q,x+32);return 0;}int crypto_scalarmult_base(u8*q,const u8*n){return crypto_scalarmult(q,n,_9);}int
crypto_box_keypair(u8*y,u8*x){randombytes(x,32);return crypto_scalarmult_base(y,x);}int crypto_box_beforenm(u8*k,const u8*y,const u8*x){u8 s
[32];crypto_scalarmult(s,x,y);return crypto_core_hsalsa20(k,_0,s,sigma);}int crypto_box_afternm(u8*c,const u8*m,u64 d,const u8*n,const u8*k)
{return crypto_secretbox(c,m,d,n,k);}int crypto_box_open_afternm(u8*m,const u8*c,u64 d,const u8*n,const u8*k){return crypto_secretbox_open(m
,c,d,n,k);}int crypto_box(u8*c,const u8*m,u64 d,const u8*n,const u8*y,const u8*x){u8 k[32];crypto_box_beforenm(k,y,x);return
crypto_box_afternm(c,m,d,n,k);}int crypto_box_open(u8*m,const u8*c,u64 d,const u8*n,const u8*y,const u8*x){u8 k[32];crypto_box_beforenm(k,y,
x);return crypto_box_open_afternm(m,c,d,n,k);}static u64 R(u64 x,int c){return(x>>c)|(x<<(64-c));}static u64 Ch(u64 x,u64 y,u64 z){return(x&
y)^(~x&z);}static u64 Maj(u64 x,u64 y,u64 z){return(x&y)^(x&z)^(y&z);}static u64 Sigma0(u64 x){return R(x,28)^R(x,34)^R(x,39);}static u64
Sigma1(u64 x){return R(x,14)^R(x,18)^R(x,41);}static u64 sigma0(u64 x){return R(x,1)^R(x,8)^(x>>7);}static u64 sigma1(u64 x){return R(x,19)^
R(x,61)^(x>>6);}static const u64 K[80]={0x428a2f98d728ae22ULL,0x7137449123ef65cdULL,0xb5c0fbcfec4d3b2fULL,0xe9b5dba58189dbbcULL,
0x3956c25bf348b538ULL,0x59f111f1b605d019ULL,0x923f82a4af194f9bULL,0xab1c5ed5da6d8118ULL,0xd807aa98a3030242ULL,0x12835b0145706fbeULL,
0x243185be4ee4b28cULL,0x550c7dc3d5ffb4e2ULL,0x72be5d74f27b896fULL,0x80deb1fe3b1696b1ULL,0x9bdc06a725c71235ULL,0xc19bf174cf692694ULL,
0xe49b69c19ef14ad2ULL,0xefbe4786384f25e3ULL,0x0fc19dc68b8cd5b5ULL,0x240ca1cc77ac9c65ULL,0x2de92c6f592b0275ULL,0x4a7484aa6ea6e483ULL,
0x5cb0a9dcbd41fbd4ULL,0x76f988da831153b5ULL,0x983e5152ee66dfabULL,0xa831c66d2db43210ULL,0xb00327c898fb213fULL,0xbf597fc7beef0ee4ULL,
0xc6e00bf33da88fc2ULL,0xd5a79147930aa725ULL,0x06ca6351e003826fULL,0x142929670a0e6e70ULL,0x27b70a8546d22ffcULL,0x2e1b21385c26c926ULL,
0x4d2c6dfc5ac42aedULL,0x53380d139d95b3dfULL,0x650a73548baf63deULL,0x766a0abb3c77b2a8ULL,0x81c2c92e47edaee6ULL,0x92722c851482353bULL,
0xa2bfe8a14cf10364ULL,0xa81a664bbc423001ULL,0xc24b8b70d0f89791ULL,0xc76c51a30654be30ULL,0xd192e819d6ef5218ULL,0xd69906245565a910ULL,
0xf40e35855771202aULL,0x106aa07032bbd1b8ULL,0x19a4c116b8d2d0c8ULL,0x1e376c085141ab53ULL,0x2748774cdf8eeb99ULL,0x34b0bcb5e19b48a8ULL,
0x391c0cb3c5c95a63ULL,0x4ed8aa4ae3418acbULL,0x5b9cca4f7763e373ULL,0x682e6ff3d6b2b8a3ULL,0x748f82ee5defb2fcULL,0x78a5636f43172f60ULL,
0x84c87814a1f0ab72ULL,0x8cc702081a6439ecULL,0x90befffa23631e28ULL,0xa4506cebde82bde9ULL,0xbef9a3f7b2c67915ULL,0xc67178f2e372532bULL,
0xca273eceea26619cULL,0xd186b8c721c0c207ULL,0xeada7dd6cde0eb1eULL,0xf57d4f7fee6ed178ULL,0x06f067aa72176fbaULL,0x0a637dc5a2c898a6ULL,
0x113f9804bef90daeULL,0x1b710b35131c471bULL,0x28db77f523047d84ULL,0x32caab7b40c72493ULL,0x3c9ebe0a15c9bebcULL,0x431d67c49c100d4cULL,
0x4cc5d4becb3e42b6ULL,0x597f299cfc657e2aULL,0x5fcb6fab3ad6faecULL,0x6c44198c4a475817ULL};int crypto_hashblocks(u8*x,const u8*m,u64 n){u64 z[
8],b[8],a[8],w[16],t;int i,j;FOR(i,8)z[i]=a[i]=dl64(x+8*i);while(n>=128){FOR(i,16)w[i]=dl64(m+8*i);FOR(i,80){FOR(j,8)b[j]=a[j];t=a[7]+Sigma1
(a[4])+Ch(a[4],a[5],a[6])+K[i]+w[i%16];b[7]=t+Sigma0(a[0])+Maj(a[0],a[1],a[2]);b[3]+=t;FOR(j,8)a[(j+1)%8]=b[j];if(i%16==15)FOR(j,16)w[j]+=w[
(j+9)%16]+sigma0(w[(j+1)%16])+sigma1(w[(j+14)%16]);}FOR(i,8)a[i]+=z[i];z[i]=a[i];}n+=128;n-=128;}FOR(i,8)ts64(x+8*i,z[i]);return n;}static
const u8 iv[64]={0x6a,0x09,0xe6,0x67,0xf3,0xbc,0xc9,0x08,0xbb,0x67,0xae,0x85,0x84,0xca,0xa7,0x3b,0x3c,0x6e,0xf3,0x72,0xfe,0x94,0xf8,0x2b,
0xa5,0x4f,0xf5,0x3a,0x5f,0x1d,0x36,0xf1,0x51,0x0e,0x52,0x7f,0xad,0xe6,0x82,0xd1,0x9b,0x05,0x68,0x8c,0x2b,0x3e,0x6c,0x1f,0x1f,0x83,0xd9,0xab,
0xfb,0x41,0xbd,0x6b,0x5b,0xe0,0xcd,0x19,0x13,0x7e,0x21,0x79};int crypto_hash(u8*out,const u8*m,u64 n){u8 h[64],x[256];u64 i,b=n;FOR(i,64)h[i
]=iv[i];crypto_hashblocks(h,m,n);m+=n;n&=127;m-=n;FOR(i,256)x[i]=0;FOR(i,n)x[i]=m[i];x[n]=128;n=256-128*(n<112);x[n-9]=b>>61;ts64(x+n-8,b<<3
);crypto_hashblocks(h,x,n);FOR(i,64)out[i]=h[i];return 0;}sv add(gf p[4],gf q[4]){gf a,b,c,d,t,e,f,g,h;Z(a,p[1],p[0]);Z(t,q[1],q[0]);M(a,a,t
);A(b,p[0],p[1]);A(t,q[0],q[1]);M(b,b,t);M(c,p[3],q[3]);M(c,c,D2);M(d,p[2],q[2]);A(d,d,d);Z(e,b,a);Z(f,d,c);A(g,d,c);A(h,b,a);M(p[0],e,f);M(
p[1],h,g);M(p[2],g,f);M(p[3],e,h);}sv cswap(gf p[4],gf q[4],u8 b){int i;FOR(i,4)sel25519(p[i],q[i],b);}sv pack(u8*r,gf p[4]){gf tx,ty,zi;
inv25519(zi,p[2]);M(tx,p[0],zi);M(ty,p[1],zi);pack25519(r,ty);r[31]^=par25519(tx)<<7;}sv scalarmult(gf p[4],gf q[4],const u8*s){int i;
set25519(p[0],gf0);set25519(p[1],gf1);set25519(p[2],gf1);set25519(p[3],gf0);for(i=255;i>=0;--i){u8 b=(s[i/8]>>(i&7))&1;cswap(p,q,b);add(q,p)
;add(p,p);cswap(p,q,b);}}sv scalarbase(gf p[4],const u8*s){gf q[4];set25519(q[0],X);set25519(q[1],Y);set25519(q[2],gf1);M(q[3],X,Y);
scalarmult(p,q,s);}int crypto_sign_keypair(u8*pk,u8*sk){u8 d[64];gf p[4];int i;randombytes(sk,32);crypto_hash(d,sk,32);d[0]&=248;d[31]&=127;
d[31]|=64;scalarbase(p,d);pack(pk,p);FOR(i,32)sk[32+i]=pk[i];return 0;}static const u64 L[32]={0xed,0xd3,0xf5,0x5c,0x1a,0x63,0x12,0x58,0xd6,
0x9c,0xf7,0xa2,0xde,0xf9,0xde,0x14,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0x10};sv modL(u8*r,i64 x[64]){i64 carry,i,j;for(i=63;i>=32;--i){carry=0;for
(j=i-32;j<i-12;++j){x[j]+=carry-16*x[i]*L[j-(i-32)];carry=(x[j]+128)>>8;x[j]-=carry<<8;}x[j]+=carry;x[i]=0;}carry=0;FOR(j,32){x[j]+=carry-(x
[31]>>4)*L[j];carry=x[j]>>8;x[j]&=255;}FOR(j,32)x[j]-=carry*L[j];FOR(i,32){x[i+1]+=x[i]>>8;r[i]=x[i]&255;}}sv reduce(u8*r){i64 x[64],i;FOR(i
,64)x[i]=(u64)r[i];FOR(i,64)r[i]=0;modL(r,x);}int crypto_sign(u8*sm,u64*smlen,const u8*m,u64 n,const u8*sk){u8 d[64],h[64],r[64];i64 i,j,x[
64];gf p[4];crypto_hash(d,sk,32);d[0]&=248;d[31]&=127;d[31]|=64;*smlen=n+64;FOR(i,n)sm[64+i]=m[i];FOR(i,32)sm[32+i]=d[32+i];crypto_hash(r,sm
+32,n+32);reduce(r);scalarbase(p,r);pack(sm,p);FOR(i,32)sm[i+32]=sk[i+32];crypto_hash(h,sm,n+64);reduce(h);FOR(i,64)x[i]=0;FOR(i,32)x[i]=(
u64)r[i];FOR(i,32)FOR(j,32)x[i+j]+=h[i]*(u64)d[j];modL(sm+32,x);return 0;}static int unpackneg(gf r[4],const u8 p[32]){gf t,chk,num,den,den2
,den4,den6;set25519(r[2],gf1);unpack25519(r[1],p);S(num,r[1]);M(den,num,D);Z(num,num,r[2]);A(den,r[2],den);S(den2,den);S(den4,den2);M(den6,
den4,den2);M(t,den6,num);M(t,t,den);pow2523(t,t);M(t,t,num);M(t,t,den);M(r[0],t,den);S(chk,r[0]);M(chk,chk,den);if(neq25519(chk,
num))M(r[0],r[0],I);S(chk,r[0]);M(chk,chk,den);if(neq25519(chk,num))return-1;if(par25519(r[0])==(p[31]>>7))Z(r[0],gf0,r[0]);M(r[3],r[0],r[1]
);return 0;}int crypto_sign_open(u8*m,u64*mlen,const u8*sm,u64 n,const u8*pk){int i;u8 t[32],h[64];gf p[4],q[4];*mlen= -1;if(n<64)return-1;
if(unpackneg(q,pk))return-1;FOR(i,n)m[i]=sm[i];FOR(i,32)m[i+32]=pk[i];crypto_hash(h,m,n);reduce(h);scalarmult(p,q,h);scalarbase(q,sm+32);add
(p,q);pack(t,p);n-=64;if(crypto_verify_32(sm,t)){FOR(i,n)m[i]=0;return-1;}FOR(i,n)m[i]=sm[i+64];*mlen=n;return 0;}
```

B A Python script to convert `tweetnacl.c` into the 100 tweets

```
import re
import sys

output = ''

while True:
  line = sys.stdin.readline()
  if not line: break
  if line[0] == '#':
    if output:
      print output
      output = ''
    print line.strip()
  else:
    x = re.findall('\w+|\W',line)
    for u in x:
      if not u.isspace():
        if len(output) + len(u) > 140:
          print output
          output = ''
        if (re.match('\w',output[-1:]) and re.match('\w',u[:1])) or (output[-1:] == '=' and u[:1] == '-'):
          if len(output) + 1 + len(u) > 140:
            print output
            output = ''
          else:
            output += ' '
        output += u

print output
```

C A Python script to print `tweetnacl.h`

```
print '#ifndef TWEETNACL_H'
print '#define TWEETNACL_H'

for z in [
'auth:hmacsha512256/32/32:BYTES,KEYBYTES:,_verify:qpup,ppup',
'box:curve25519xsalsa20poly1305/32/32/32/24/32/16:PUBLICKEYBYTES,SECRETKEYBYTES,BEFORENMBYTES,NONCEBYTES,ZEROBYTES,BOXZEROBYTES:'
+ ',_open,_keypair,_beforenm,_afternm,_open_afternm:qpuppp,qpuppp,qq,qpp,qpupp,qpupp',
'core:salsa20/64/16/32/16,hsalsa20/32/16/32/16:OUTPUTBYTES,INPUTBYTES,KEYBYTES,CONSTBYTES::qppp',
'hashblocks:sha512/64/128,sha256/32/64:STATEBYTES,BLOCKBYTES::qpu',
'hash:sha512/64,sha256/32:BYTES::qpu',
'onetimeauth:poly1305/16/32:BYTES,KEYBYTES:,_verify:qpup,ppup',
'scalarmult:curve25519/32/32:BYTES,SCALARBYTES:,_base:qpp,qp',
'secretbox:xsalsa20poly1305/32/24/32/16:KEYBYTES,NONCEBYTES,ZEROBYTES,BOXZEROBYTES:,_open:qpupp,qpupp',
'sign:ed25519/64/32/64:BYTES,PUBLICKEYBYTES,SECRETKEYBYTES:,_open,_keypair:qvpup,qvpup,qq',
'stream:xsalsa20/32/24,salsa20/32/8:KEYBYTES,NONCEBYTES:,_xor:qupp,qpupp',
'verify:16/16,32/32:BYTES::pp'
]:
  x,q,s,f,g = [i.split(',') for i in z.split(':')]
  o = 'crypto_'+x[0]
  sel = 1
  for p in q:
    p = p.split('/')
    op = o+'_'+p[0]
    opi = op+'_'+'tweet'
    if sel:
      print '#define '+o+'_PRIMITIVE "'+p[0]+'"'
      for m in f+['_'+m for m in s+['IMPLEMENTATION','VERSION']]: print '#define '+o+m+' '+op+m
      sel = 0
    for j in range(len(s)): print '#define '+opi+'_'+s[j]+' '+str(p[j+1])
    for j in range(len(f)):
      a = g[j].replace('v','u *').replace('u',',unsigned long long').replace('q',',unsigned char *').replace('p',',const unsigned char *')
      print 'extern int '+opi+f[j]+'('+a[1:]+');'
    print '#define '+opi+'_VERSION "-"'
    for m in f+['_'+m for m in s+['VERSION']]: print '#define '+op+m+' '+opi+m
    print '#define '+op+'_IMPLEMENTATION "'+o+'/'+p[0]+'/tweet'+'"'

print '#endif'
```

D Table of symbols

name	type	meaning
`_0`	`const u8[16]`	`{0}`
`_9`	`const u8[32]`	`{9}`
`_121665`	`const gf`	`{0xDB41,1}`
`A`	function	add 256-bit integers, radix 2^{16}
`add`	function	add points on Edwards curve
`add1305`	function	add 136-bit integers, radix 2^8
`car25519`	function	reduce mod $2^{255}-19$, radix 2^{16}
`Ch(x,y,z)`	function	`((x & y) ^ (~x & z))`

`core`	function	merged `crypto_core_salsa20`, `crypto_core_hsalsa20`
`cswap`	function	conditionally swap curve points
`D`	`const gf`	Edwards curve parameter
`D2`	`const gf`	Edwards curve parameter, doubled
`dl64`	function	load 64-bit integer big-endian
`FOR(i,n)`	macro	`for (i = 0;i < n;++i)`
`gf`	`typedef`	`i64 [16]`, representing 256-bit integer in radix 2^{16}
`gf0`	`const gf`	`{0}`
`gf1`	`const gf`	`{1}`
`I`	`const gf`	$\sqrt{-1} \bmod 2^{255} - 19$
`i64`	typedef	signed $\geq$64-bit integer (`long long`)
`inv25519`	function	power $2^{255} - 21 \bmod 2^{255} - 19$
`iv`	`const u8[64]`	initialization vector for SHA-512
`K`	`const u64[80]`	constants for SHA-512
`L`	`const u64[32]`	prime order of base point
`L32`	function	rotate 32-bit integer left
`ld32`	function	load 32-bit integer little-endian
`M`	function	multiply mod $2^{255} - 19$, radix 2^{16}
`Maj(x,y,z)`	function	`((x & y) ^ (x & z) ^ (y & z))`
`minusp`	`const u32[17]`	`{5,0,...,0,252}`
`modL`	function	freeze mod order of base point, radix 2^8
`neq25519`	function	compare mod $2^{255} - 19$
`pack`	function	freeze and store curve point
`pack25519`	function	freeze integer mod $2^{255} - 19$ and store
`par25519`	function	parity of integer mod $2^{255} - 19$
`pow2523`	function	power $2^{252} - 3 \bmod 2^{255} - 19$
`R`	function	rotate 64-bit integer right
`reduce`	function	freeze 512-bit string mod order of base point
`S`	function	square mod $2^{255} - 19$, radix 2^{16}
`scalarbase`	function	scalar multiplication by base point on Edwards curve
`scalarmult`	function	scalar multiplication on Edwards curve
`sel25519`	function	256-bit conditional swap
`set25519`	function	copy 256-bit integer
`sigma`	`const u8[16]`	Salsa20 constant: `"expand 32-byte k"`
`sigma0(x)`	function	`(R(x, 1) ^ R(x, 8) ^ (x >> 7))`
`Sigma0(x)`	function	`(R(x,28) ^ R(x,34) ^ R(x,39))`
`sigma1(x)`	function	`(R(x,19) ^ R(x,61) ^ (x >> 6))`
`Sigma1(x)`	function	`(R(x,14) ^ R(x,18) ^ R(x,41))`
`st32`	function	store 32-bit integer little-endian
`sv`	macro	`static void`
`ts64`	function	store 64-bit integer big-endian
`u8`	typedef	unsigned 8-bit integer (`unsigned char`)
`u32`	typedef	unsigned $\geq$32-bit integer (`unsigned long`)
`u64`	typedef	unsigned $\geq$64-bit integer (`unsigned long long`)
`unpack25519`	function	load integer mod $2^{255} - 19$
`unpackneg`	function	load curve point
`vn`	function	merged `crypto_verify_16`, `crypto_verify_32`

`X`	`const gf`	x-coordinate of base point
`Y`	`const gf`	y-coordinate of base point
`Z`	function	subtract 256-bit integers, radix 2^{16}