CACE

Computer Aided Cryptography Engineering

Project number: 216499

FP7-ICT-2007-1

D2.5
Improved Networking and Cryptography Library

Due date of deliverable: 31. December 2010
Actual submission date: 21. February 2011

WP contributing to the deliverable: WP2

Start date of project: 1. January 2008 Duration: 3 years

Coordinator:
Technikon Forschungs- und Planungsgesellschaft mbH
Burgplatz 3a, 9500 Villach, Austria
Phone: +43 4242 233 550
Email: coordination@cace-project.eu
www.cace-project.eu

Revision 1.0

Project co-funded by the European Commission within the 7th Framework Programme		
Dissemination Level		
PU	Public	X
PP	Restricted to other programme participants (including the Commission services)	
RE	Restricted to a group specified by the consortium (including the Commission services)	
CO	Confidential, only for members of the consortium (including the Commission services)	

Improved Networking and Cryptography Library

Editor
Tanja Lange (TUE)

Contributors
Daniel J. Bernstein (UIC)
Peter Schwabe (TUE)

21. February 2011
Revision 1.0

Abstract

This report describes the functionality, installation, and use of the Networking and Cryptography library (NaCl).

Keywords: NaCl, networking, cryptography

Contents

Part I:

NaCl for programmers

Chapter 1

Introduction to NaCl

"NaCl" (pronounced "salt") is the CACE Networking and Cryptography library, a new easy-to-use high-speed high-security public-domain software library for network communication, encryption, decryption, signatures, etc. NaCl's goal is to provide all of the core operations needed to build higher-level cryptographic tools.

Of course, other libraries already exist for these core operations. NaCl advances the state of the art by improving security, by improving usability, and by improving speed.

1.1 Contributors

NaCl was initiated by the CACE (Computer Aided Cryptography Engineering) project funded by the European Commission's Seventh Framework Programme (FP7), contract number ICT-2008-216499. CACE activities were organized into several Work Packages (WPs). NaCl was the main task of CACE WP2, "Accelerating Secure Networking," led by Tanja Lange (at Technische Universiteit Eindhoven) and Daniel J. Bernstein (at the University of Illinois at Chicago, currently visiting Eindhoven). CACE finished at the end of 2010 but NaCl is a continuing project.

NaCl benefits from close collaboration with two other projects. The NaCl API is based on, and has influenced, the SUPERCOP (System for Unified Performance Evaluation Related to Cryptographic Operations and Primitives) API developed for the eBACS (ECRYPT Benchmarking of Cryptographic Systems) project [18]. Many of the algorithms used in NaCl were developed as part of Daniel J. Bernstein's High-Speed Cryptography project funded by the U.S. National Science Foundation, grant number ITR-0716498. "Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation."

Emilia Käsper at Katholieke Universiteit Leuven and Peter Schwabe at Technische Universiteit Eindhoven wrote the fast AES implementations used in C NaCl. Peter Schwabe also implemented signatures for C NaCl and C++ NaCl.

Adam Langley at Google wrote the `donna` and `donna_c64` implementations of Curve25519 in C NaCl and a prototype Python NaCl wrapper around C NaCl.

Matthew Dempsky at Mochi Media wrote the `ref` implementation of Curve25519 in C NaCl and reference implementations of many functions in Python NaCl.

Chapter 2

Features

2.1 C NaCl, C++ NaCl, and Python NaCl

The current version of NaCl supports C and C++. Support for Python is a high priority.

NaCl takes advantage of higher-level language features to simplify the APIs for those languages. For example:

- A message is represented in C NaCl as two variables: an array variable `m` and an integer variable `mlen`. Higher-level APIs use a single string variable `m` that knows its own length.
- The C NaCl functions return error codes to indicate, e.g., invalid signatures; a caller that neglects to check for those error codes will blithely proceed as if the signatures were valid. Higher-level APIs raise exceptions.
- The C NaCl functions write output strings via pointers. Higher-level APIs simply return the strings as function values.

2.2 High-speed implementations of high-speed primitives

NaCl aims to, and in many cases already does, provide record-setting speeds for each of its cryptographic operations.

This means not merely record-setting speeds for (e.g.) bulk data encryption with AES, but record-setting speeds for bulk data encryption with the best secret-key cipher available. Note that a state-of-the-art stream cipher such as Salsa20 is considerably faster than AES.

Sometimes NaCl includes slow implementations of primitives that are expected to set speed records with better implementations. Applications built using these NaCl functions will eventually benefit from higher-speed implementations.

2.3 Automatic CPU-specific tuning

NaCl, like other speed-oriented cryptographic libraries, supports multiple implementations of the same function.

Most of these libraries attempt to recognize CPUs and select implementations (and compiler options) known to work well on those CPUs. NaCl instead compiles all implementations (with all specified compiler options) and uses speed measurements to select the fastest implementation for the user's CPU.

One advantage of NaCl's automatic tuning is that new CPUs automatically use the best option available, without any human intervention. Another advantage is that new implementations (and new compiler options) can be added with a minimum of fuss.

In theory, there is a compile-time cost for automatic tuning, and the cost grows with the number of implementations (and compiler options). In practice, the cost of NaCl's automatic tuning is quite small, and if it ever becomes troublesome then it can easily be split across many machines.

2.4 Support for standard primitives

Whenever NaCl includes–for speed reasons or for security reasons–a newly proposed cipher, a newly proposed signature system, etc., it also includes an older standard cipher (e.g., AES), [TO DO:] an older standard signature system (e.g., ECDSA using the NIST P-256 elliptic curve), etc. Some users avoid new proposals as a matter of policy; NaCl accommodates those users.

This does not mean that it is a high priority for NaCl to support *every* cryptographic standard. Example: An implementation of triple DES might be of interest for bank-industry applications, but is a lower priority for NaCl than an improved implementation of AES.

2.5 Expert selection of default primitives

Typical cryptographic libraries force the programmer to specify choices of cryptographic primitives: e.g., "sign this message with 4096-bit RSA using PKCS #1 v2.0 with SHA-256."

Most programmers using cryptographic libraries are not expert cryptographic security evaluators. Often programmers pass the choice along to users–who usually have even less information about the security of cryptographic primitives. There is a long history of these programmers and users making poor choices of cryptographic primitives, such as MD5 and 512-bit RSA, years after cryptographers began issuing warnings about the security of those primitives.

NaCl allows, and encourages, the programmer to simply say "sign this message." NaCl has a side mechanism through which a cryptographer can easily specify the choice of signature system. Furthermore, NaCl is shipped with a preselected choice, namely a state-of-the-art signature system suitable for worldwide use in a wide range of applications.

2.6 High-level primitives

A typical cryptographic library requires several steps to authenticate and encrypt a message. Consider, for example, the following typical combination of RSA, AES, etc.:

- Generate a random AES key.
- Use the AES key to encrypt the message.
- Hash the encrypted message using SHA-256.

- Read the sender's RSA secret key from "wire format."
- Use the sender's RSA secret key to sign the hash.
- Read the recipient's RSA public key from wire format.
- Use the recipient's public key to encrypt the AES key, hash, and signature.
- Convert the encrypted key, hash, and signature to wire format.
- Concatenate with the encrypted message.

Sometimes even more steps are required for storage allocation, error handling, etc.

NaCl provides a simple `crypto_box` function that does everything in one step. The function takes the sender's secret key, the recipient's public key, and a message, and produces an authenticated ciphertext. All objects are represented in wire format, as sequences of bytes suitable for transmission; the `crypto_box` function automatically handles all necessary conversions, initializations, etc.

Another virtue of NaCl's high-level API is that it is not tied to the traditional hash-sign-encrypt-etc. hybrid structure. NaCl supports much faster message-boxing solutions that reuse Diffie-Hellman shared secrets for any number of messages between the same parties.

A multiple-step procedure can have important speed advantages when multiple computations share precomputations. NaCl allows users to split `crypto_box` into two steps, namely `crypto_box_beforenm` for message-independent precomputation and `crypto_box_afternm` for message-dependent computation.

2.7 No branches depending on secret data

The CPU's instruction pointer, branch predictor, etc. are not designed to keep information secret. For performance reasons this situation is unlikely to change. The literature has many examples of successful timing attacks that extracted secret keys from these parts of the CPU.

NaCl systematically avoids all data flow from secret information to the instruction pointer and the branch predictor. There are no conditional branches with conditions based on secret information; in particular, all loop counts are predictable in advance.

This protection appears to be compatible with extremely high speed, so there is no reason to consider weaker protections.

2.8 No array indices depending on secret data

The CPU's cache, TLB, etc. are not designed to keep addresses secret. For performance reasons this situation is unlikely to change. The literature has several examples of successful cache-timing attacks that used secret information leaked through addresses.

NaCl systematically avoids all data flow from secret information to the addresses used in load instructions and store instructions. There are no array lookups with indices based on secret information; the pattern of memory access is predictable in advance.

The conventional wisdom for many years was that achieving acceptable software speed for AES required variable-index array lookups, exposing the AES key to side-channel attacks, specifically cache-timing attacks. However, the paper "Faster and timing-attack resistant AES-GCM" by Emilia Käsper and Peter Schwabe at CHES 2009 introduced a new implementation that set record-setting speeds for AES on the popular Core 2 CPU despite being immune to cache-timing attacks. NaCl reuses these results.

2.9 No dynamic memory allocation

C NaCl is intended to be usable in environments that cannot guarantee the availability of large amounts of heap storage but that nevertheless rely on their cryptographic computations to continue working. C NaCl functions do not call `malloc`, `sbrk`, etc. They do use small amounts of stack space; these amounts will eventually be measured by separate benchmarks.

This feature applies only to C NaCl. Higher-level languages such as Python are not currently usable in restricted environments.

2.10 No copyright restrictions

All of the NaCl software is in the public domain.

Chapter 3

Installation

NaCl works on a wide variety of UNIX-like systems, including Linux, BSD, Solaris, etc. Here is how to download and compile NaCl:

```
wget http://hyperelliptic.org/nacl/nacl-20110221.tar.bz2
bunzip2 < nacl-20110221.tar.bz2 | tar -xf -
cd nacl-20110221
./do
```

Several machines sharing the same filesystem can compile NaCl simultaneously in the same shared directory. All compilation takes place in a host-specific subdirectory: for example, `build/atlas` on a machine named `atlas`, and `build/katana` on a machine named `katana`.

Some machines support multiple application binary interfaces (ABIs). For example, the machine `katana` can run 32-bit x86 programs and 64-bit amd64 programs; both types of programs are supported by the CPU (an Intel Core 2 Duo), the operating system (64-bit Ubuntu), and the compiler (gcc). NaCl automatically compiles itself for each ABI: inside `build/katana`, NaCl creates both

- a 64-bit `lib/amd64/libnacl.a` accompanied by header files `include/amd64/crypto_auth.h`, `include/amd64/crypto_stream.h`, etc., and
- a 32-bit `lib/x86/libnacl.a` accompanied by `include/x86/crypto_auth.h`, `include/x86/crypto_stream.h`, etc.

The NaCl header files are created dynamically as part of the compilation process, reflecting machine-specific optimizations, and can vary from one ABI to another.

NaCl creates an ABI-independent program `bin/okabi` that prints the list of supported ABIs: for example, `amd64` and `x86`. NaCl also creates `bin/okc-amd64` to print the list of C compilers supporting the `amd64` ABI, `bin/okc-x86` to print the list of C compilers supporting the `x86` ABI, etc.

[TO DO:] NaCl's header files and compiled code can be installed in appropriate system locations so that applications can include `<crypto_auth.h>`, link with `-lnacl`, etc. Beware that current operating systems have many incompatible, and generally deficient, mechanisms of handling multiple ABIs, making multiple-ABI support rather difficult for libraries and for applications.

Rather than relying on the system to have an installation of NaCl, applications can include the NaCl source code as part of the application source code. The applications can then use `okabi`, `okc-amd64`, etc. to automatically try all ABIs and compilers and to select the fastest option.

Chapter 4

Internals

Are you contributing code to NaCl? Here's some advice on how to write your code.

4.1 Language choice

All NaCl functions are *distributed* as C implementations, often with assembly-language implementations as extra options. C++ and Python are handled as wrappers around the C implementations.

There are an increasing number of cases where the C implementations and assembly-language implementations are automatically generated from code that was actually written in another language, such as CAO or qhasm. This automatic generation takes place as part of creating the NaCl distribution, not as part of installing NaCl.

4.2 Naming conventions: an example

The reference SHA-512 implementation in NaCl is `crypto_hash/sha512/ref/hash.c`, which defines a `crypto_hash` function as follows:

```
#include "crypto_hash.h"

int crypto_hash(unsigned char *out,
  const unsigned char *in,unsigned long long inlen)
{
  ...
}
```

An accompanying file `crypto_hash/sha512/ref/api.h` says

```
#define CRYPTO_BYTES 64
```

to indicate that this function produces a 64-byte hash.

The NaCl compilation scripts automatically create `crypto_hash.h` with various macros, including the following:

```
#define crypto_hash crypto_hash_sha512
#define crypto_hash_sha512 crypto_hash_sha512_ref
```

```
#define crypto_hash_BYTES crypto_hash_sha512_BYTES
#define crypto_hash_sha512_BYTES crypto_hash_sha512_ref_BYTES
#define crypto_hash_sha512_ref_BYTES 64
#define crypto_hash_PRIMITIVE "sha512"
#define crypto_hash_IMPLEMENTATION crypto_hash_sha512_IMPLEMENTATION
#define crypto_hash_sha512_IMPLEMENTATION "crypto_hash/sha512/ref"
```

This means that `hash.c` actually defines a `crypto_hash_sha512_ref` function. Normally, as an implementor, you don't have to think about this; but if you avoid including `crypto_hash.h` (for example, because you're actually writing `hash.s` instead of `hash.c`) then you should make sure to define `crypto_hash_sha512_ref`.

4.3 Multiple implementations of the same primitive

To add another SHA-512 implementation, create another subdirectory of `crypto_hash/sha512`. For example, you might choose `crypto_hash/sha512/core2` for an implementation optimized for the Core 2. Inside that subdirectory, create an `api.h` file that defines `CRYPTO_BYTES`, and a `.c` file that defines `crypto_hash`.

You can use names other than `hash.c`. You can split your code across several files `*.c` defining various auxiliary functions; the files will be automatically compiled together. You can use external names prefixed by the implementation name: for example, `crypto_hash/sha512/core2` can define `crypto_hash_sha512_core2_iv`, `crypto_hash_sha512_core2_expand`, etc.

4.4 Branches

Do not use secret data to control a branch. In particular, do not use the `memcmp` function to compare secrets. Instead use `crypto_verify_16`, `crypto_verify_32`, etc., which perform constant-time string comparisons.

4.5 Array lookups

Do not use secret data as an array index.

Early plans for NaCl would have allowed exceptions to this rule inside primitives specifically labelled `vulnerable`, in particular to allow fast `crypto_stream_aes128vulnerable`, but subsequent research showed that this compromise was unnecessary.

4.6 Dynamic memory allocation

Do not use `malloc`, `calloc`, `sbrk`, etc. in C NaCl.

4.7 Thread safety

Do not use global variables (i.e., static variables or variables defined outside functions) in C NaCl.

4.8 Alignment

Do not assume that the input arrays or output arrays have any particular alignment. If you want to use, e.g., an aligned 16-byte load instruction for speed, check at the top of your function that the input array is properly aligned, and have a (slower) fallback in case the input array is not properly aligned.

4.9 Specific integer sizes

If you want a 32-bit integer, use `crypto_int32` after `#include <crypto_int32.h>`. This is more portable than `int32_t` from `inttypes.h`, and more portable than `int32_t` from `stdint.h`. Similar comments apply to `crypto_int8`, `crypto_int16`, `crypto_int32`, `crypto_int64`, `crypto_uint8`, `crypto_uint16`, `crypto_uint32`, `crypto_uint64`.

Chapter 5

Validation and verification

It is essential for cryptographic libraries to compute exactly the functions that they are meant to compute, and for those functions to be secure. A signature-checking library is a security disaster if it has a bug that accepts invalid signatures, for example, or if the signature system that it implements is 512-bit RSA.

Part II of this document specifies NaCl's default mechanism for public-key authenticated encryption, and along the way specifies NaCl's default mechanisms for scalar multiplication (Curve25519), secret-key authenticated encryption, secret-key encryption (Salsa20), and one-time authentication (Poly1305). The same part includes a complete step-by-step example of authenticated encryption, independent implementations testing each step, detailed security notes, and references to the relevant literature.

The NaCl compilation scripts test known outputs of each primitive for many different message lengths, test consistency of different functions supported by the same primitive (for example, `crypto_stream_xor` matches `crypto_stream`), and test memory safety in several ways.

Chapter 6

Public-key authenticated encryption: crypto_box

6.1 C++ interface

C++ NaCl provides a crypto_box_keypair function callable as follows:

```
#include "crypto_box.h"

std::string pk;
std::string sk;

pk = crypto_box_keypair(&sk);
```

The crypto_box_keypair function randomly generates a secret key and a corresponding public key. It puts the secret key into sk and returns the public key. It guarantees that sk has crypto_box_SECRETKEYBYTES bytes and that pk has crypto_box_PUBLICKEYBYTES bytes.

C++ NaCl also provides a crypto_box function callable as follows:

```
#include "crypto_box.h"

std::string pk;
std::string sk;
std::string n;
std::string m;
std::string c;

c = crypto_box(m,n,pk,sk);
```

The crypto_box function encrypts and authenticates a message m using the sender's secret key sk, the receiver's public key pk, and a nonce n. The crypto_box function returns the resulting ciphertext c. The function raises an exception if sk.size() is not crypto_box_SECRETKEYBYTES or if pk.size() is not crypto_box_PUBLICKEYBYTES or if n.size() is not crypto_box_NONCEBYTES.

C++ NaCl also provides a crypto_box_open function callable as follows:

```
#include "crypto_box.h"

std::string pk;
std::string sk;
std::string n;
std::string c;
std::string m;

m = crypto_box_open(c,n,pk,sk);
```

The `crypto_box_open` function verifies and decrypts a ciphertext `c` using the receiver's secret key `sk`, the sender's public key `pk`, and a nonce `n`. The `crypto_box_open` function returns the resulting plaintext `m`.

If the ciphertext fails verification, `crypto_box_open` raises an exception. The function also raises an exception if `sk.size()` is not `crypto_box_SECRETKEYBYTES` or if `pk.size()` is not `crypto_box_PUBLICKEYBYTES` or if `n.size()` is not `crypto_box_NONCEBYTES`.

6.2 C interface

C NaCl provides a `crypto_box_keypair` function callable as follows:

```
#include "crypto_box.h"

unsigned char pk[crypto_box_PUBLICKEYBYTES];
unsigned char sk[crypto_box_SECRETKEYBYTES];

crypto_box_keypair(pk,sk);
```

The `crypto_box_keypair` function randomly generates a secret key and a corresponding public key. It puts the secret key into `sk[0]`, `sk[1]`, ..., `sk[crypto_box_SECRETKEYBYTES-1]` and puts the public key into `pk[0]`, `pk[1]`, ..., `pk[crypto_box_PUBLICKEYBYTES-1]`. It then returns 0.

C NaCl also provides a `crypto_box` function callable as follows:

```
#include "crypto_box.h"

const unsigned char pk[crypto_box_PUBLICKEYBYTES];
const unsigned char sk[crypto_box_SECRETKEYBYTES];
const unsigned char n[crypto_box_NONCEBYTES];
const unsigned char m[...]; unsigned long long mlen;
unsigned char c[...];

crypto_box(c,m,mlen,n,pk,sk);
```

The `crypto_box` function encrypts and authenticates a message `m[0]`, ..., `m[mlen-1]` using the sender's secret key `sk[0]`, `sk[1]`, ..., `sk[crypto_box_SECRETKEYBYTES-1]`, the receiver's public key `pk[0]`, `pk[1]`, ..., `pk[crypto_box_PUBLICKEYBYTES-1]`, and a nonce `n[0]`, `n[1]`, ..., `n[crypto_box_NONCEBYTES-1]`. The `crypto_box` function puts the ciphertext into `c[0]`, `c[1]`, ..., `c[mlen-1]`. It then returns 0.

WARNING: Messages in the C NaCl API are 0-padded versions of messages in the C++ NaCl API. Specifically: The caller must ensure, before calling the C NaCl `crypto_box` function, that the first `crypto_box_ZEROBYTES` bytes of the message `m` are all 0. Typical higher-level applications will work with the remaining bytes of the message; note, however, that `mlen` counts all of the bytes, including the bytes required to be 0.

Similarly, ciphertexts in the C NaCl API are 0-padded versions of messages in the C++ NaCl API. Specifically: The `crypto_box` function ensures that the first `crypto_box_BOXZEROBYTES` bytes of the ciphertext `c` are all 0.

C NaCl also provides a `crypto_box_open` function callable as follows:

```
#include "crypto_box.h"

const unsigned char pk[crypto_box_PUBLICKEYBYTES];
const unsigned char sk[crypto_box_SECRETKEYBYTES];
const unsigned char n[crypto_box_NONCEBYTES];
const unsigned char c[...]; unsigned long long clen;
unsigned char m[...];

crypto_box_open(m,c,clen,n,pk,sk);
```

The `crypto_box_open` function verifies and decrypts a ciphertext `c[0]`, ..., `c[clen-1]` using the receiver's secret key `sk[0]`, `sk[1]`, ..., `sk[crypto_box_SECRETKEYBYTES-1]`, the sender's public key `pk[0]`, `pk[1]`, ..., `pk[crypto_box_PUBLICKEYBYTES-1]`, and a nonce `n[0]`, ..., `n[crypto_box_NONCEBYTES-1]`. The `crypto_box_open` function puts the plaintext into `m[0]`, `m[1]`, ..., `m[clen-1]`. It then returns 0.

If the ciphertext fails verification, `crypto_box_open` instead returns `-1`, possibly after modifying `m[0]`, `m[1]`, etc.

The caller must ensure, before calling the `crypto_box_open` function, that the first `crypto_box_BOXZEROBYTES` bytes of the ciphertext `c` are all 0. The `crypto_box_open` function ensures (in case of success) that the first `crypto_box_ZEROBYTES` bytes of the plaintext `m` are all 0.

6.3 C precomputation interface

Applications that send several messages to the same receiver can gain speed by splitting `crypto_box` into two steps, `crypto_box_beforenm` and `crypto_box_afternm`. Similarly, applications that receive several messages from the same sender can gain speed by splitting `crypto_box_open` into two steps, `crypto_box_beforenm` and `crypto_box_open_afternm`.

The `crypto_box_beforenm` function is callable as follows:

```
#include "crypto_box.h"

unsigned char k[crypto_box_BEFORENMBYTES];
const unsigned char pk[crypto_box_PUBLICKEYBYTES];
const unsigned char sk[crypto_box_SECRETKEYBYTES];

crypto_box_beforenm(k,pk,sk);
```

The crypto_box_afternm function is callable as follows:

```
#include "crypto_box.h"

const unsigned char k[crypto_box_BEFORENMBYTES];
const unsigned char n[crypto_box_NONCEBYTES];
const unsigned char m[...]; unsigned long long mlen;
unsigned char c[...];

crypto_box_afternm(c,m,mlen,n,k);
```

The crypto_box_open_afternm function is callable as follows:

```
#include "crypto_box.h"

const unsigned char k[crypto_box_BEFORENMBYTES];
const unsigned char n[crypto_box_NONCEBYTES];
const unsigned char c[...]; unsigned long long clen;
unsigned char m[...];

crypto_box_open_afternm(m,c,clen,n,k);
```

The intermediate data computed by crypto_box_beforenm is suitable for both crypto_box_afternm and crypto_box_open_afternm, and can be reused for any number of messages.

6.4 Security model

The crypto_box function is designed to meet the standard notions of privacy and third-party unforgeability for a public-key authenticated-encryption scheme using nonces. For formal definitions see, e.g., [2].

Distinct messages between the same {sender, receiver} set are required to have distinct nonces. For example, the lexicographically smaller public key can use nonce 1 for its first message to the other key, nonce 3 for its second message, nonce 5 for its third message, etc., while the lexicographically larger public key uses nonce 2 for its first message to the other key, nonce 4 for its second message, nonce 6 for its third message, etc. Nonces are long enough that randomly generated nonces have negligible risk of collision.

There is no harm in having the same nonce for different messages if the {sender, receiver} sets are different. This is true even if the sets overlap. For example, a sender can use the same nonce for two different messages if the messages are sent to two different public keys.

The crypto_box function is not meant to provide non-repudiation. On the contrary: the crypto_box function *guarantees* repudiability. A receiver can freely modify a boxed message, and therefore cannot convince third parties that this particular message came from the sender. The sender and receiver are nevertheless protected against forgeries by other parties. In the terminology of [9], crypto_box uses "public-key authenticators" rather than "public-key signatures."

Users who want public verifiability (or receiver-assisted public verifiability) should instead use signatures (or signcryption).

6.5 Selected primitive

`crypto_box` is `curve25519xsalsa20poly1305`, a particular combination of Curve25519, Salsa20, and Poly1305 specified in Part II of this document. This function is conjectured to meet the standard notions of privacy and third-party unforgeability.

6.6 Alternate primitives

NaCl supports the following public-key message-protection functions:

	`crypto_box_...BYTES`					
`crypto_box`	`PUBLICKEY`	`SECRETKEY`	`NONCE`	`ZERO`	`BOXZERO`	`BEFORENM`
[TO DO:] `crypto_box_nistp256aes256gcm`	64	32	8	32	0	32
`crypto_box_curve25519xsalsa20poly1305`	32	32	24	32	16	32

For example, a user can replace `crypto_box` etc. with `crypto_box_curve25519xsalsa20poly1305` etc.

Chapter 7

Scalar multiplication: crypto_scalarmult

7.1 C++ interface

C++ NaCl provides a `crypto_scalarmult` function callable as follows:

```
#include "crypto_scalarmult.h"

std::string p;
std::string n;
std::string q;

q = crypto_scalarmult(n,p);
```

This function multiplies a group element `p` by an integer `n`. It returns the resulting group element `q` of length `crypto_scalarmult_BYTES`. The function raises an exception if `p.size()` is not `crypto_scalarmult_BYTES`. It also raises an exception if `n.size()` is not `crypto_scalarmult_SCALARBYTES`.

C++ NaCl also provides a `crypto_scalarmult_base` function callable as follows:

```
#include "crypto_scalarmult.h"

std::string n;
std::string q;

q = crypto_scalarmult_base(n);
```

The `crypto_scalarmult_base` function computes the scalar product of a standard group element and an integer `n`. It returns the resulting group element `q` of length `crypto_scalarmult_BYTES`. It raises an exception if `n.size()` is not `crypto_scalarmult_SCALARBYTES`.

7.2 C interface

C NaCl provides a `crypto_scalarmult` function callable as follows:

```
#include "crypto_scalarmult.h"

const unsigned char p[crypto_scalarmult_BYTES];
const unsigned char n[crypto_scalarmult_SCALARBYTES];
unsigned char q[crypto_scalarmult_BYTES];

crypto_scalarmult(q,n,p);
```

This function multiplies a group element `p[0]`, ..., `p[crypto_scalarmult_BYTES-1]` by an integer `n[0]`, ..., `n[crypto_scalarmult_SCALARBYTES-1]`. It puts the resulting group element into `q[0]`, ..., `q[crypto_scalarmult_BYTES-1]` and returns 0.

C NaCl also provides a `crypto_scalarmult_base` function callable as follows:

```
#include "crypto_scalarmult.h"

const unsigned char n[crypto_scalarmult_SCALARBYTES];
unsigned char q[crypto_scalarmult_BYTES];

crypto_scalarmult_base(q,n);
```

The `crypto_scalarmult_base` function computes the scalar product of a standard group element and an integer `n[0]`, ..., `n[crypto_scalarmult_SCALARBYTES-1]`. It puts the resulting group element into `q[0]`, ..., `q[crypto_scalarmult_BYTES-1]` and returns 0.

7.3 Representation of group elements

The correspondence between strings and group elements depends on the primitive implemented by `crypto_scalarmult`. The correspondence is not necessarily injective in either direction, but it is compatible with scalar multiplication in the group. The correspondence does not necessarily include all group elements, but it *does* include all strings; i.e., every string represents at least one group element.

7.4 Representation of integers

The correspondence between strings and integers also depends on the primitive implemented by `crypto_scalarmult`. Every string represents at least one integer.

7.5 Security model

`crypto_scalarmult` is designed to be strong as a component of various well-known "hashed Diffie–Hellman" applications. In particular, it is designed to make the "computational Diffie–Hellman" problem (CDH) difficult with respect to the standard base.

`crypto_scalarmult` is also designed to make CDH difficult with respect to other nontrivial bases. In particular, if a represented group element has small order, then it is annihilated by *all* represented scalars. This feature allows protocols to avoid validating membership in the subgroup generated by the standard base.

NaCl does not make any promises regarding the “decisional Diffie–Hellman” problem (DDH), the “static Diffie–Hellman” problem (SDH), etc. Users are responsible for hashing group elements.

7.6 Selected primitive

`crypto_scalarmult` is the function `crypto_scalarmult_curve25519` specified in Part II of this document. This function is conjectured to be strong. For background see [13].

7.7 Alternate primitives

NaCl supports the following scalar-multiplication functions:

`crypto_scalarmult`	`BYTES`	`SCALARBYTES`
[TO DO:] `crypto_scalarmult_nistp256`	64	32
`crypto_scalarmult_curve25519`	32	32

For example, a user who wants to use the Curve25519 group can replace `crypto_scalarmult`, `crypto_scalarmult_BYTES`, etc. with `crypto_scalarmult_curve25519`, `crypto_scalarmult_curve25519_BYTES`, etc.

Chapter 8

Secret-key authenticated encryption: crypto_secretbox

8.1 C++ interface

C++ NaCl provides a `crypto_secretbox` function callable as follows:

```
#include "crypto_secretbox.h"

std::string k;
std::string n;
std::string m;
std::string c;

c = crypto_secretbox(m,n,k);
```

The `crypto_secretbox` function encrypts and authenticates a message `m` using a secret key `k` and a nonce `n`. The `crypto_secretbox` function returns the resulting ciphertext `c`. The function raises an exception if `k.size()` is not `crypto_secretbox_KEYBYTES`. The function also raises an exception if `n.size()` is not `crypto_secretbox_NONCEBYTES`.

C++ NaCl also provides a `crypto_secretbox_open` function callable as follows:

```
#include "crypto_secretbox.h"

std::string k;
std::string n;
std::string c;
std::string m;

m = crypto_secretbox_open(c,n,k);
```

The `crypto_secretbox_open` function verifies and decrypts a ciphertext `c` using a secret key `k` and a nonce `n`. The `crypto_secretbox_open` function returns the resulting plaintext `m`.

If the ciphertext fails verification, `crypto_secretbox_open` raises an exception. The function also raises an exception if `k.size()` is not `crypto_secretbox_KEYBYTES`, or if `n.size()` is not `crypto_secretbox_NONCEBYTES`.

8.2 C interface

C NaCl provides a `crypto_secretbox` function callable as follows:

```
#include "crypto_secretbox.h"

const unsigned char k[crypto_secretbox_KEYBYTES];
const unsigned char n[crypto_secretbox_NONCEBYTES];
const unsigned char m[...]; unsigned long long mlen;
unsigned char c[...]; unsigned long long clen;

crypto_secretbox(c,m,mlen,n,k);
```

The `crypto_secretbox` function encrypts and authenticates a message `m[0]`, `m[1]`, ..., `m[mlen-1]` using a secret key `k[0]`, ..., `k[crypto_secretbox_KEYBYTES-1]` and a nonce `n[0]`, `n[1]`, ..., `n[crypto_secretbox_NONCEBYTES-1]`. The `crypto_secretbox` function puts the ciphertext into `c[0]`, `c[1]`, ..., `c[mlen-1]`. It then returns 0.

WARNING: Messages in the C NaCl API are 0-padded versions of messages in the C++ NaCl API. Specifically: The caller must ensure, before calling the C NaCl `crypto_secretbox` function, that the first `crypto_secretbox_ZEROBYTES` bytes of the message `m` are all 0. Typical higher-level applications will work with the remaining bytes of the message; note, however, that `mlen` counts all of the bytes, including the bytes required to be 0.

Similarly, ciphertexts in the C NaCl API are 0-padded versions of messages in the C++ NaCl API. Specifically: The `crypto_secretbox` function ensures that the first `crypto_secretbox_BOXZEROBYTES` bytes of the ciphertext `c` are all 0.

C NaCl also provides a `crypto_secretbox_open` function callable as follows:

```
#include "crypto_secretbox.h"

const unsigned char k[crypto_secretbox_KEYBYTES];
const unsigned char n[crypto_secretbox_NONCEBYTES];
const unsigned char c[...]; unsigned long long clen;
unsigned char m[...];

crypto_secretbox_open(m,c,clen,n,k);
```

The `crypto_secretbox_open` function verifies and decrypts a ciphertext `c[0]`, `c[1]`, ..., `c[clen-1]` using a secret key `k[0]`, `k[1]`, ..., `k[crypto_secretbox_KEYBYTES-1]` and a nonce `n[0]`, ..., `n[crypto_secretbox_NONCEBYTES-1]`. The `crypto_secretbox_open` function puts the plaintext into `m[0]`, `m[1]`, ..., `m[clen-1]`. It then returns 0.

If the ciphertext fails verification, `crypto_secretbox_open` instead returns `-1`, possibly after modifying `m[0]`, `m[1]`, etc.

The caller must ensure, before calling the `crypto_secretbox_open` function, that the first `crypto_secretbox_BOXZEROBYTES` bytes of the ciphertext `c` are all 0. The `crypto_secretbox_open` function ensures (in case of success) that the first `crypto_secretbox_ZEROBYTES` bytes of the plaintext `m` are all 0.

8.3 Security model

The crypto_secretbox function is designed to meet the standard notions of privacy and authenticity for a secret-key authenticated-encryption scheme using nonces. For formal definitions see, e.g., [8].

Note that the length is not hidden. Note also that it is the caller's responsibility to ensure the uniqueness of nonces—for example, by using nonce 1 for the first message, nonce 2 for the second message, etc. Nonces are long enough that randomly generated nonces have negligible risk of collision.

8.4 Selected primitive

crypto_secretbox is crypto_secretbox_xsalsa20poly1305, a particular combination of Salsa20 and Poly1305 specified in Part II of this document. This function is conjectured to meet the standard notions of privacy and authenticity.

8.5 Alternate primitives

NaCl supports the following secret-key message-protection functions:

crypto_secretbox	KEYBYTES	NONCEBYTES	ZEROBYTES	BOXZEROBYTES
[TO DO:] crypto_secretbox_aes256gcm	32	8	32	0
crypto_secretbox_xsalsa20poly1305	32	24	32	16

For example, a user who wants to encrypt and authenticate messages with AES-256-GCM can replace crypto_secretbox with crypto_secretbox_aes256gcm, crypto_secretbox_KEYBYTES with crypto_secretbox_aes256gcm_KEYBYTES, etc.

Beware that some of these primitives have 8-byte nonces. For those primitives it is no longer true that randomly generated nonces have negligible risk of collision. Callers who are unable to count 1,2,3,..., and who insist on using these primitives, are advised to use a randomly derived key for each message.

Chapter 9

Secret-key encryption: crypto_stream

9.1 C++ interface

C++ NaCl provides a crypto_stream function callable as follows:

```
#include "crypto_stream.h"

std::string k;
std::string n;
std::string c;
size_t clen;

c = crypto_stream(clen,n,k);
```

The crypto_stream function produces a clen-byte stream c as a function of a secret key k and a nonce n. The function raises an exception if k.size() is not crypto_stream_KEYBYTES. It also raises an exception if n.size() is not crypto_stream_NONCEBYTES.

C++ NaCl also provides a crypto_stream_xor function callable as follows:

```
#include "crypto_stream.h"

std::string k;
std::string n;
std::string m;
std::string c;

c = crypto_stream_xor(m,n,k);
```

The crypto_stream_xor function encrypts a message m using a secret key k and a nonce n. The crypto_stream_xor function returns the ciphertext c. The function raises an exception if k.size() is not crypto_stream_KEYBYTES. It also raises an exception if n.size() is not crypto_stream_NONCEBYTES.

The crypto_stream_xor function guarantees that the ciphertext has the same length as the plaintext, and is the plaintext xor the output of crypto_stream. Consequently crypto_stream_xor can also be used to decrypt.

9.2 C interface

C NaCl provides a `crypto_stream` function callable as follows:

```
#include "crypto_stream.h"

const unsigned char k[crypto_stream_KEYBYTES];
const unsigned char n[crypto_stream_NONCEBYTES];
unsigned char c[...]; unsigned long long clen;

crypto_stream(c,clen,n,k);
```

The `crypto_stream` function produces a stream `c[0]`, `c[1]`, ..., `c[clen-1]` as a function of a secret key `k[0]`, `k[1]`, ..., `k[crypto_stream_KEYBYTES-1]` and a nonce `n[0]`, `n[1]`, ..., `n[crypto_stream_NONCEBYTES-1]`. The `crypto_stream` function then returns 0.

C NaCl also provides a `crypto_stream_xor` function callable as follows:

```
#include "crypto_stream.h"

const unsigned char k[crypto_stream_KEYBYTES];
const unsigned char n[crypto_stream_NONCEBYTES];
unsigned char m[...]; unsigned long long mlen;
unsigned char c[...];

crypto_stream_xor(c,m,mlen,n,k);
```

The `crypto_stream_xor` function encrypts a message `m[0]`, `m[1]`, ..., `m[mlen-1]` using a secret key `k[0]`, `k[1]`, ..., `k[crypto_stream_KEYBYTES-1]` and a nonce `n[0]`, `n[1]`, ..., `n[crypto_stream_NONCEBYTES-1]`. The `crypto_stream_xor` function puts the ciphertext into `c[0]`, `c[1]`, ..., `c[mlen-1]`. It then returns 0.

The `crypto_stream_xor` function guarantees that the ciphertext is the plaintext xor the output of `crypto_stream`. Consequently `crypto_stream_xor` can also be used to decrypt.

9.3 Security model

The `crypto_stream` function, viewed as a function of the nonce for a uniform random key, is designed to meet the standard notion of unpredictability ("PRF"). For a formal definition see, e.g., [7, Section 2.3].

This means that an attacker cannot distinguish this function from a uniform random function. Consequently, if a series of messages is encrypted by `crypto_stream_xor` *with a different nonce for each message*, the ciphertexts are indistinguishable from uniform random strings of the same length.

Note that the length is not hidden. Note also that it is the caller's responsibility to ensure the uniqueness of nonces—for example, by using nonce 1 for the first message, nonce 2 for the second message, etc. Nonces are long enough that randomly generated nonces have negligible risk of collision.

NaCl does not make any promises regarding the resistance of `crypto_stream` to "related-key attacks." It is the caller's responsibility to use proper key-derivation functions.

9.4 Selected primitive

`crypto_stream` is `crypto_stream_xsalsa20`, a particular cipher specified in Part II of this document. This cipher is conjectured to meet the standard notion of unpredictability.

9.5 Alternate primitives

NaCl supports the following secret-key encryption functions:

`crypto_stream`	Primitive	`KEYBYTES`	`NONCEBYTES`
`crypto_stream_aes128ctr`	AES-128-CTR	16	16
[TO DO:] `crypto_stream_aes256ctr`	AES-256-CTR	32	16
`crypto_stream_salsa208`	Salsa20/8	32	8
`crypto_stream_salsa2012`	Salsa20/12	32	8
`crypto_stream_salsa20`	Salsa20/20	32	8
`crypto_stream_xsalsa20`	XSalsa20/20	32	24

For example, a user who wants to encrypt with AES-128 can replace `crypto_stream`, `crypto_stream_KEYBYTES`, etc. with `crypto_stream_aes128ctr`, `crypto_stream_aes128ctr_KEYBYTES`, etc.

Beware that several of these primitives have 8-byte nonces. For those primitives it is no longer true that randomly generated nonces have negligible risk of collision. Callers who are unable to count 1,2,3,..., and who insist on using these primitives, are advised to use a randomly derived key for each message.

Chapter 10

Secret-key message authentication: crypto_auth

10.1 C++ interface

C++ NaCl provides a crypto_auth function callable as follows:

```
#include "crypto_auth.h"

std::string k;
std::string m;
std::string a;

a = crypto_auth(m,k);
```

The crypto_auth function authenticates a message m using a secret key k. The function returns an authenticator a. The authenticator length is always crypto_auth_BYTES. The function raises an exception if k.size() is not crypto_auth_KEYBYTES.

C++ NaCl also provides a crypto_auth_verify function callable as follows:

```
#include "crypto_auth.h"

std::string k;
std::string m;
std::string a;

crypto_auth_verify(a,m,k);
```

The crypto_auth_verify function checks that k.size() is crypto_auth_KEYBYTES; a.size() is crypto_auth_BYTES; and a is a correct authenticator of a message m under the secret key k. If any of these checks fail, the function raises an exception.

10.2 C interface

C NaCl provides a crypto_auth function callable as follows:

```
#include "crypto_auth.h"

const unsigned char k[crypto_auth_KEYBYTES];
const unsigned char m[...]; unsigned long long mlen;
unsigned char a[crypto_auth_BYTES];

crypto_auth(a,m,mlen,k);
```

The `crypto_auth` function authenticates a message `m[0]`, `m[1]`, ..., `m[mlen-1]` using a secret key `k[0]`, `k[1]`, ..., `k[crypto_auth_KEYBYTES-1]`. The `crypto_auth` function puts the authenticator into `a[0]`, `a[1]`, ..., `a[crypto_auth_BYTES-1]`. It then returns 0.

C NaCl also provides a `crypto_auth_verify` function callable as follows:

```
#include "crypto_auth.h"

const unsigned char k[crypto_auth_KEYBYTES];
const unsigned char m[...]; unsigned long long mlen;
const unsigned char a[crypto_auth_BYTES];

crypto_auth_verify(a,m,mlen,k);
```

The `crypto_auth_verify` function returns `0` if `a[0]`, ..., `a[crypto_auth_BYTES-1]` is a correct authenticator of a message `m[0]`, `m[1]`, ..., `m[mlen-1]` under a secret key `k[0]`, `k[1]`, ..., `k[crypto_auth_KEYBYTES-1]`. Otherwise `crypto_auth_verify` returns `-1`.

10.3 Security model

The `crypto_auth` function, viewed as a function of the message for a uniform random key, is designed to meet the standard notion of unforgeability. This means that an attacker cannot find authenticators for any messages not authenticated by the sender, even if the attacker has adaptively influenced the messages authenticated by the sender. For a formal definition see, e.g., [7, Section 2.4].

NaCl does not make any promises regarding "strong" unforgeability; perhaps one valid authenticator can be converted into another valid authenticator for the same message. NaCl also does not make any promises regarding "truncated unforgeability."

10.4 Selected primitive

`crypto_auth` is currently an implementation of HMAC-SHA-512-256, i.e., the first 256 bits of HMAC-SHA-512. HMAC-SHA-512-256 is conjectured to meet the standard notion of unforgeability.

10.5 Alternate primitives

NaCl supports the following secret-key authentication functions:

crypto_auth	Primitive	BYTES	KEYBYTES
crypto_auth_hmacsha256	HMAC-SHA-256	32	32
crypto_auth_hmacsha512256	HMAC-SHA-512-256	32	32

For example, a user can replace **crypto_auth**, **crypto_auth_KEYBYTES**, etc. with **crypto_auth_hmacsha256**, **crypto_auth_hmacsha256_KEYBYTES**, etc.

Chapter 11

Secret-key single-message authentication: crypto_onetimeauth

11.1 C++ interface

C++ NaCl provides a `crypto_onetimeauth` function callable as follows:

```
#include "crypto_onetimeauth.h"

std::string k;
std::string m;
std::string a;

a = crypto_onetimeauth(m,k);
```

The `crypto_onetimeauth` function authenticates a message `m` using a secret key `k`, and returns an authenticator `a`. The authenticator length is always `crypto_onetimeauth_BYTES`. The function raises an exception if `k.size()` is not `crypto_onetimeauth_KEYBYTES`.

C++ NaCl also provides a `crypto_onetimeauth_verify` function callable as follows:

```
#include "crypto_onetimeauth.h"

std::string k;
std::string m;
std::string a;

crypto_onetimeauth_verify(a,m,k);
```

This function checks that `k.size()` is `crypto_onetimeauth_KEYBYTES`; `a.size()` is `crypto_onetimeauth_BYTES`; and `a` is a correct authenticator of a message `m` under the secret key `k`. If any of these checks fail, the function raises an exception.

11.2 C interface

C NaCl provides a `crypto_onetimeauth` function callable as follows:

```
#include "crypto_onetimeauth.h"

const unsigned char k[crypto_onetimeauth_KEYBYTES];
const unsigned char m[...]; unsigned long long mlen;
unsigned char a[crypto_onetimeauth_BYTES];

crypto_onetimeauth(a,m,mlen,k);
```

The `crypto_onetimeauth` function authenticates a message `m[0]`, `m[1]`, ..., `m[mlen-1]` using a secret key `k[0]`, `k[1]`, ..., `k[crypto_onetimeauth_KEYBYTES-1]`; puts the authenticator into `a[0]`, `a[1]`, ..., `a[crypto_onetimeauth_BYTES-1]`; and returns 0.

C NaCl also provides a `crypto_onetimeauth_verify` function callable as follows:

```
#include "crypto_onetimeauth.h"

const unsigned char k[crypto_onetimeauth_KEYBYTES];
const unsigned char m[...]; unsigned long long mlen;
const unsigned char a[crypto_onetimeauth_BYTES];

crypto_onetimeauth_verify(a,m,mlen,k);
```

This function returns `0` if `a[0]`, `a[1]`, ..., `a[crypto_onetimeauth_BYTES-1]` is a correct authenticator of a message `m[0]`, `m[1]`, ..., `m[mlen-1]` under a secret key `k[0]`, `k[1]`, ..., `k[crypto_onetimeauth_KEYBYTES-1]`. Otherwise `crypto_onetimeauth_verify` returns `-1`.

11.3 Security model

The `crypto_onetimeauth` function, viewed as a function of the message for a uniform random key, is designed to meet the standard notion of unforgeability after a single message. After the sender authenticates *one* message, an attacker cannot find authenticators for any other messages.

The sender must not use `crypto_onetimeauth` to authenticate more than one message under the same key. Authenticators for two messages under the same key should be expected to reveal enough information to allow forgeries of authenticators on other messages.

11.4 Selected primitive

`crypto_onetimeauth` is `crypto_onetimeauth_poly1305`, an authenticator specified in Part II of this document. This authenticator is *proven* to meet the standard notion of unforgeability after a single message.

11.5 Alternate primitives

NaCl supports the following secret-key single-message authentication functions:

crypto_onetimeauth	Primitive	BYTES	KEYBYTES
crypto_onetimeauth_poly1305	Poly1305	16	32

For example, a user can replace `crypto_onetimeauth`, `crypto_onetimeauth_BYTES`, etc. with `crypto_onetimeauth_poly1305`, `crypto_onetimeauth_poly1305_BYTES`, etc. Furthermore, users willing to compromise both provability and speed can replace `crypto_onetimeauth` with `crypto_auth` or with any of the `crypto_auth` primitives.

Chapter 12

Hashing: crypto_hash

12.1 C++ interface

C++ NaCl provides a `crypto_hash` function callable as follows:

```
#include "crypto_hash.h"

std::string m;
std::string h;

h = crypto_hash(m);
```

The `crypto_hash` function hashes a message `m`. It returns a hash `h`. The output length `h.size()` is always `crypto_hash_BYTES`.

12.2 C interface

C NaCl provides a `crypto_hash` function callable as follows:

```
#include "crypto_hash.h"

const unsigned char m[...]; unsigned long long mlen;
unsigned char h[crypto_hash_BYTES];

crypto_hash(h,m,mlen);
```

The `crypto_hash` function hashes a message `m[0]`, `m[1]`, ..., `m[mlen-1]`. It puts the hash into `h[0]`, `h[1]`, ..., `h[crypto_hash_BYTES-1]`. It then returns 0.

12.3 Security model

The `crypto_hash` function is designed to be usable as a strong component of DSA, RSA-PSS, key derivation, hash-based message-authentication codes, hash-based ciphers, and various other common applications. "Strong" means that the security of these applications, when instantiated with `crypto_hash`, is the same as the security of the applications against generic attacks. In particular, the `crypto_hash` function is designed to make finding collisions difficult.

12.4 Selected primitive

`crypto_hash` is currently an implementation of SHA-512.

There has been considerable degradation of public confidence in the security conjectures for many hash functions, including SHA-512. However, for the moment, there do not appear to be alternatives that inspire satisfactory levels of confidence. One can hope that NIST's SHA-3 competition will improve the situation.

12.5 Alternate primitives

NaCl supports the following hash functions:

`crypto_hash`	Primitive	`BYTES`
`crypto_hash_sha256`	SHA-256	32
`crypto_hash_sha512`	SHA-512	64

For example, a user who wants to hash with SHA-256 can simply replace `crypto_hash`, `crypto_hash_BYTES`, etc. with `crypto_hash_sha256`, `crypto_hash_sha256_BYTES`, etc.

Chapter 13

String comparison: crypto_verify

C NaCl provides a `crypto_verify_16` function callable as follows:

```
#include "crypto_verify_16.h"

const unsigned char x[16];
const unsigned char y[16];

crypto_verify_16(x,y);
```

The `crypto_verify_16` function returns `0` if `x[0]`, `x[1]`, ..., `x[15]` are the same as `y[0]`, `y[1]`, ..., `y[15]`. Otherwise it returns `-1`.

This function is safe to use for secrets `x[0]`, `x[1]`, ..., `x[15]`, `y[0]`, `y[1]`, ..., `y[15]`. The time taken by `crypto_verify_16` is independent of the contents of `x[0]`, `x[1]`, ..., `x[15]`, `y[0]`, `y[1]`, ..., `y[15]`. In contrast, the standard C comparison function `memcmp(x,y,16)` takes time that depends on the longest matching prefix of `x` and `y`, often allowing easy timing attacks.

C NaCl also provides a similar `crypto_verify_32` function.

Chapter 14

Signatures: crypto_sign

14.1 C++ interface

C++ NaCl provides a `crypto_sign_keypair` function callable as follows:

```
#include "crypto_sign.h"

std::string pk;
std::string sk;

pk = crypto_sign_keypair(&sk);
```

The `crypto_sign_keypair` function randomly generates a secret key and a corresponding public key. It puts the secret key into `sk` and returns the public key. It guarantees that `sk` has `crypto_sign_SECRETKEYBYTES` bytes and that `pk` has `crypto_sign_PUBLICKEYBYTES` bytes.

C++ NaCl also provides a `crypto_sign` function callable as follows:

```
#include "crypto_sign.h"

std::string sk;
std::string m;
std::string sm;

sm = crypto_sign(m,sk);
```

The `crypto_sign` function signs a message `m` using the sender's secret key `sk`. The `crypto_sign` function returns the resulting signed message `sm`. The function raises an exception if `sk.size()` is not `crypto_sign_SECRETKEYBYTES`.

C++ NaCl also provides a `crypto_sign_open` function callable as follows:

```
#include "crypto_sign.h"

std::string pk;
std::string sm;
std::string m;

m = crypto_sign_open(sm,pk);
```

The crypto_sign_open function verifies the signature in sm using the receiver's secret key sk. The crypto_sign_open function returns the message m.

If the signature fails verification, crypto_sign_open raises an exception. The function also raises an exception or if pk.size() is not crypto_sign_PUBLICKEYBYTES.

14.2 C interface

C NaCl provides a crypto_sign_keypair function callable as follows:

```
#include "crypto_sign.h"

unsigned char pk[crypto_sign_PUBLICKEYBYTES];
unsigned char sk[crypto_sign_SECRETKEYBYTES];

crypto_sign_keypair(pk,sk);
```

The crypto_sign_keypair function randomly generates a secret key and a corresponding public key. It puts the secret key into sk[0], sk[1], ..., sk[crypto_sign_SECRETKEYBYTES-1] and puts the public key into pk[0], pk[1], ..., pk[crypto_sign_PUBLICKEYBYTES-1]. It then returns 0.

C NaCl also provides a crypto_sign function callable as follows:

```
#include "crypto_sign.h"

const unsigned char sk[crypto_sign_SECRETKEYBYTES];
const unsigned char m[...]; unsigned long long mlen;
unsigned char sm[...]; unsigned long long smlen;

crypto_sign(sm,&smlen,m,mlen,sk);
```

The crypto_sign function encrypts and authenticates a message m[0], ..., m[mlen-1] using the sender's secret key sk[0], sk[1], ..., sk[crypto_sign_SECRETKEYBYTES-1], puts the length of the signed message into smlen and puts the signed message into sm[0], sm[1], ..., sm[smlen-1]. It then returns 0.

WARNING: The maximal length of the signed message sm is mlen+crypto_sign_BYTES. The caller must allocate at least mlen+crypto_sign_BYTES bytes for sm.

C NaCl also provides a crypto_sign_open function callable as follows:

```
#include "crypto_sign.h"

const unsigned char pk[crypto_sign_PUBLICKEYBYTES];
const unsigned char sm[...]; unsigned long long smlen;
unsigned char m[...]; unsigned long long mlen;

crypto_sign_open(m,&mlen,sm,smlen,pk);
```

The crypto_sign_open function verifies the signature in sm[0], ..., sm[smlen-1] using the receiver's public key pk[0], pk[1], ..., pk[crypto_sign_PUBLICKEYBYTES-1], The crypto_sign_open function puts the length of the message into mlen and puts the message into m[0], m[1], ..., m[mlen-1]. It then returns 0. The caller has to ensure that smlen bytes are allocated for m.

If the signature fails verification, crypto_sign_open instead returns -1, possibly after modifying m[0], m[1], etc.

14.3 Security model

The crypto_sign function is designed to meet the standard notion of unforgeability for a public-key signature scheme under chosen-message attacks.

14.4 Selected primitive

crypto_sign is crypto_sign_edwards25519sha512batch, a particular combination of Curve25519 in Edwards form and SHA-512 into a signature scheme suitable for high-speed batch verification. See Chapter 25 for more details. This function is conjectured to meet the standard notion of unforgeability under chosen-message attacks; it is also conjectured to meet the standard notion of non-malleability.

14.5 Alternate primitives

NaCl supports the following public-key signature functions:

	crypto_sign_...		
crypto_sign	PUBLICKEYBYTES	SECRETKEYBYTES	BYTES
[TO DO:] crypto_sign_nistp256sha512ecdsa	64	64	64
crypto_sign_edwards25519sha512batch	32	64	64

For example, a user who wants to encrypt and authenticate messages with the NIST P-256 curve using SHA-512 and the ECDSA algorithm can replace crypto_sign with crypto_sign_nistp256sha512ecdsa.

Part II:

NaCl for cryptographers

Chapter 15

Overview of cryptography in NaCl

The following chapters comprehensively document the cryptography used in NaCl.

The most fundamental operation in a cryptographically protected network protocol is **public-key authenticated encryption**. The sender, Alice, has a packet of data to send to the receiver, Bob. Alice scrambles the packet using her own secret key and Bob's public key. Bob unscrambles the packet using his secret key and Alice's public key. An attacker monitoring the network is unable to understand the scrambled packet; an attacker modifying network packets is unable to change the packet produced by Bob's unscrambling.

With typical cryptographic libraries, public-key authenticated encryption takes several steps. Here is a typical series of steps:

- Generate a random AES key.
- Use the AES key to encrypt the packet.
- Hash the encrypted packet using SHA-256.
- Read Alice's RSA secret key from "wire format."
- Use Alice's RSA secret key to sign the hash.
- Read Bob's RSA public key from wire format.
- Use Bob's public key to encrypt the AES key, hash, and signature.
- Convert the encrypted key, hash, and signature to wire format.
- Concatenate with the encrypted packet.

NaCl provides a high-level function `crypto_box` that does everything in one step, converting a packet into a boxed packet that is protected against espionage and sabotage. Programmers can use lower-level functions but are encouraged to use `crypto_box`.

In particular, `crypto_box_curve25519xsalsa20poly1305` is a specific high-speed high-security combination of the Curve25519 elliptic-curve-Diffie–Hellman function, the Salsa20 stream cipher, and the Poly1305 message-authentication code. This combination is designed for universal use and is shipped in NaCl as the default definition of `crypto_box`.

The following chapters specify exactly what this combination does: i.e., exactly how the boxed packet produced by `crypto_box_curve25519xsalsa20poly1305` relates to the inputs.

The specification is expressed as a step-by-step procedure for Alice to encrypt and authenticate a packet; NaCl might compute the boxed packet in a different way but produces exactly the same results. Three of the steps are packet-independent precomputation:

- Chapter 16: Alice creates a 32-byte secret key a and a 32-byte public key A. These keys can be reused for other packets to Bob, for packets to other receivers, and for packets sent back from the receivers.
- Chapter 16, continued: Bob creates a 32-byte secret key b and a 32-byte public key B. These keys can be reused for other packets from Alice, for packets from other senders, and for packets sent back to the senders.
- Chapter 19: Alice, using Alice's secret key a and Bob's public key B, computes a 32-byte secret k. Bob can compute the same secret using Bob's secret key b and Alice's public key A.

The remaining three steps are specific to one packet:

- Chapter 21: Alice, using a 24-byte nonce (unique packet number) n that will never be reused for other packets to (or from) Bob, expands the shared secret k into a long stream of secret bytes. Bob, given the nonce, can compute the same stream.
- Chapter 23: Alice uses the long stream, except for the first 32 bytes, to encrypt the packet m.
- Chapter 23, continued: Alice uses the first 32 bytes of the long stream to compute an authenticator of the encrypted packet.

Each chapter includes security notes and pointers to the relevant literature.

This part also contains, in Chapters 17, 18, 20, 22, and 24, a complete step-by-step example to illustrate the specification. The intermediate results are printed by various C NaCl programs shown here.

This part also contains several tests showing that C NaCl is producing the same results as independent programs in other languages. Some of the tests rely on scripts using the Sage computer-algebra system [32], and some of the tests rely on Python scripts contributed by Matthew Dempsky. This part can be used as a starting point for more comprehensive NaCl validation and verification.

In these chapters, a **byte** is an element of $\{0, 1, \ldots, 255\}$. NaCl works with all keys, packets, etc. as strings of bytes. For example, the set of 32-byte strings is the set $\{0, 1, \ldots, 255\}^{32}$.

Chapter 16

Secret keys and public keys

Alice's secret key is a string $a \in \{0, 1, \ldots, 255\}^{32}$. Alice's public key is a string Curve25519(ClampC(a), $\underline{9}$) $\in \{0, 1, \ldots, 255\}^{32}$. Similarly, Bob's secret key is a string $b \in \{0, 1, \ldots, 255\}^{32}$, and Bob's public key is Curve25519(ClampC(b), $\underline{9}$) $\in \{0, 1, \ldots, 255\}^{32}$.

This chapter defines the functions ClampC and Curve25519 and the constant $\underline{9}$. Many of the definitions here are copied from [13, Section 2]; in particular, Curve25519 here is the same as the Curve25519 function defined in [13].

Chapter 17 gives an example of a secret key and corresponding public key that Alice might choose. Chapter 18 gives an example of a secret key and corresponding public key that Bob might choose. These examples are reused in subsequent chapters.

The base field and the elliptic curve. Define $p = 2^{255} - 19$. This integer is prime:

```
    sage: p=2^255-19
    sage: p.is_prime()
    True
```

Define $\mathbf{F}_p$ as the prime field $\mathbf{Z}/p = \mathbf{Z}/(2^{255} - 19)$. Note that 2 is not a square in $\mathbf{F}_p$:

```
    sage: p=2^255-19
    sage: k=GF(p)
    sage: k(2).is_square()
    False
```

Define $\mathbf{F}_{p^2}$ as the field $(\mathbf{Z}/(2^{255} - 19))[\sqrt{2}]$. Define $a_2 = 486662$. Note that $a_2^2 - 4$ is not a square in $\mathbf{F}_p$:

```
    sage: p=2^255-19
    sage: k=GF(p)
    sage: a2=486662
    sage: (k(a2)^2-4).is_square()
    False
```

Define E as the elliptic curve $y^2 = x^3 + a_2x^2 + x$ over $\mathbf{F}_p$, and define $E(\mathbf{F}_{p^2})$ as the group of points of E with coordinates in $\mathbf{F}_{p^2}$. Readers not familiar with elliptic curves can find a self-contained definition of $E(\mathbf{F}_{p^2})$ in [13, Appendix A].

Define $X_0 : E(\mathbf{F}_{p^2}) \to \mathbf{F}_{p^2}$ as follows: $X_0(\infty) = 0$; $X_0(x, y) = x$.

The Curve25519 function. Write $s \mapsto \underline{s}$ for the standard little-endian bijection from $\{0, 1, \ldots, 2^{256} - 1\}$ to $\{0, 1, \ldots, 255\}^{32}$. In other words, for each integer $s \in \{0, 1, \ldots, 2^{256} - 1\}$, define

$$\underline{s} = (s \bmod 256, \lfloor s/256 \rfloor \bmod 256, \ldots, \lfloor s/256^{31} \rfloor \bmod 256).$$

For example, the constant $\underline{9}$ is $(9, 0, 0, \ldots, 0) \in \{0, 1, \ldots, 255\}^{32}$.

The set of **Curve25519 secret keys** is, by definition, $\{0, 8, 16, 24, \ldots, 248\} \times \{0, 1, \ldots, 255\}^{30} \times \{64, 65, 66, \ldots, 127\}$. If $n \in 2^{254} + 8\{0, 1, 2, 3, \ldots, 2^{251} - 1\}$ then $\underline{n}$ is a Curve25519 secret key; and every Curve25519 secret key can be written as $\underline{n}$ for a unique $n \in 2^{254} + 8\{0, 1, 2, 3, \ldots, 2^{251} - 1\}$.

Now the function

$$\text{Curve25519} : \{\text{Curve25519 secret keys}\} \times \{0, 1, \ldots, 255\}^{32} \to \{0, 1, \ldots, 255\}^{32}$$

is defined as follows. Fix an integer $n \in 2^{254} + 8\{0, 1, 2, 3, \ldots, 2^{251} - 1\}$ and an integer $q \in \{0, 1, \ldots, 2^{256} - 1\}$. By [13, Theorem 2.1] there is a unique integer $s \in \{0, 1, \ldots, 2^{255} - 20\}$ such that $X_0(nQ) = s$ for all $Q \in E(\mathbf{F}_{p^2})$ such that $X_0(Q) = q \bmod 2^{255} - 19$. Finally, Curve25519$(\underline{n}, \underline{q})$ is defined as $\underline{s}$.

The ClampC function. The function

$$\text{ClampC} : \{0, 1, \ldots, 255\}^{32} \to \{\text{Curve25519 secret keys}\}$$

maps $(a_0, a_1, \ldots, a_{30}, a_{31})$ to $(a_0 - (a_0 \bmod 8), a_1, \ldots, a_{30}, 64 + (a_{31} \bmod 64))$. In other words, ClampC clears bits $(7, 0, \ldots, 0, 0, 128)$ and sets bit $(0, 0, \ldots, 0, 0, 64)$.

Specialization of Curve25519 for secret keys. Note that $9^3 + a_2 \cdot 9^2 + 9 = 39420360$:

```
sage: a2=486662
sage: 9^3+a2*9^2+9
39420360
```

If $n \in 2^{254} + 8\{0, 1, 2, 3, \ldots, 2^{251} - 1\}$ then Curve25519$(\underline{n}, \underline{9}) = \underline{s}$, where s is the unique integer in $\{0, 1, \ldots, 2^{255} - 20\}$ such that $X_0(n(9, \pm\sqrt{39420360})) = s$. Consequently, if Alice's secret key a satisfies ClampC$(a) = \underline{n}$, then Alice's public key is $\underline{s}$.

The range of n implies that $n(9, \pm\sqrt{39420360}) \neq \infty$, so ∞ could be omitted from the definition of X_0 for purposes of computing secret keys. However, Alice also applies Curve25519 to network inputs, as discussed in subsequent chapters, and there are several ways that attacker-chosen inputs can lead to the ∞ case.

ECDLP security notes. The following notes assume additional familiarity with elliptic curves.

Write $Q = (9, \sqrt{39420360})$. The choice of square root is not relevant here. This point Q is in the subgroup $E(\mathbf{F}_p)$ of $E(\mathbf{F}_{p^2})$:

```
sage: p=2^255-19
sage: k=GF(p)
sage: k(39420360).is_square()
True
```

Furthermore, Q has p_1th multiple equal to ∞ in $E(\mathbf{F}_p)$, where p_1 is the prime number $2^{252} +$ 27742317777372353535851937790883648493:

```
sage: p=2^255-19
sage: k=GF(p)
sage: p1=2^252+27742317777372353535851937790883648493
sage: p1.is_prime()
True
sage: E=EllipticCurve([k(0),486662,0,1,0])
sage: Q=[k(9),sqrt(k(39420360))]
sage: p1*E(Q)
(0 : 1 : 0)
```

Consequently all multiples of Q are in the subgroup of $E(\mathbf{F}_p)$ of order p_1.

If Alice's secret key a is a uniform random 32-byte string then $\text{ClampC}(a)$ is a uniform random Curve25519 secret key; i.e., $\underline{n}$, where $n/8$ is a uniform random integer between 2^{251} and $2^{252} - 1$. Alice's public key is nQ compressed to the x-coordinate (as recommended in [28, page 425, fourth paragraph] in 1986). Note that n is not a multiple of p_1; this justifies the statement above that $nQ \neq \infty$.

The problem of finding Alice's secret key from Alice's public key is exactly the elliptic-curve 251-bit-discrete-logarithm problem for the subgroup of $E(\mathbf{F}_p)$ of order $p_1 \approx 2^{252}$. The curve E meets all of the standard security criteria, as discussed in detail in [13, Section 3]. The fastest known attacks use, on average, about 2^{125} additions in $E(\mathbf{F}_p)$, and have success chance degrading quadratically as the number of additions decreases.

It is standard in the literature to restrict attention to uniform random secret keys. What if the key distribution is not uniform? The answer depends on the distribution. For example, a key derived from an 8-byte string can be found by brute-force search in roughly 2^{64} operations; and a key derived in an *extremely weak* way from a 16-byte string, for example by concatenating a 16-byte public constant, can also be found in roughly 2^{64} operations. On the other hand, it is easy to prove that *slightly* non-uniform keys have essentially full security. Furthermore, we are not aware of any feasible attacks against 32-byte keys of the form $(s, \text{MD5}(s))$, where s is a uniform random 16-byte string; none of the weaknesses of MD5 seem relevant here. Constructions of this type allow secret-key compression and might merit further study if there are any applications where memory is filled with secret keys.

Chapter 17

Example of the sender's keys

The following program uses C NaCl to compute the public key corresponding to a particular secret key:

```
#include <stdio.h>
#include "crypto_scalarmult_curve25519.h"

unsigned char alicesk[32] = {
 0x77,0x07,0x6d,0x0a,0x73,0x18,0xa5,0x7d
,0x3c,0x16,0xc1,0x72,0x51,0xb2,0x66,0x45
,0xdf,0x4c,0x2f,0x87,0xeb,0xc0,0x99,0x2a
,0xb1,0x77,0xfb,0xa5,0x1d,0xb9,0x2c,0x2a
} ;

unsigned char alicepk[32];

main()
{
  int i;
  crypto_scalarmult_curve25519_base(alicepk,alicesk);
  for (i = 0;i < 32;++i) {
    if (i > 0) printf(","); else printf(" ");
    printf("0x%02x",(unsigned int) alicepk[i]);
    if (i % 8 == 7) printf("\n");
  }
  return 0;
}
```

The secret key bytes `0xc7,0x6e,...` embedded into the program were copied from output of `od -t x1 /dev/urandom` head -2—. The output of the program is the corresponding public key:

```
 0x85,0x20,0xf0,0x09,0x89,0x30,0xa7,0x54
,0x74,0x8b,0x7d,0xdc,0xb4,0x3e,0xf7,0x5a
,0x0d,0xbf,0x3a,0x0d,0x26,0x38,0x1a,0xf4
,0xeb,0xa4,0xa9,0x8e,0xaa,0x9b,0x4e,0x6a
```

The remaining chapters of Part II will reuse this example, assuming that Alice's keys are the particular secret key and public key shown here.

Testing: Sage vs. `scalarmult_curve25519_base`. A short Sage script clamps Alice's secret key shown above, converts the result to an integer n, computes $n(9, \sqrt{39420360})$ in $E(\mathbf{F}_p)$, and checks that the resulting x-coordinate matches the public key computed by C NaCl:

```
sage: sk=[0x77,0x07,0x6d,0x0a,0x73,0x18,0xa5,0x7d
....:    ,0x3c,0x16,0xc1,0x72,0x51,0xb2,0x66,0x45
....:    ,0xdf,0x4c,0x2f,0x87,0xeb,0xc0,0x99,0x2a
....:    ,0xb1,0x77,0xfb,0xa5,0x1d,0xb9,0x2c,0x2a]
sage: clampsk=sk
sage: clampsk[0]=clampsk[0]-(clampsk[0]%8)
sage: clampsk[31]=64+(clampsk[31]%64)
sage: n=sum(clampsk[i]*256^i for i in range(32))
sage: p=2^255-19
sage: k=GF(p)
sage: E=EllipticCurve([k(0),486662,0,1,0])
sage: s=lift((n*E([k(9),sqrt(k(39420360))]))[0])
sage: pk=[0x85,0x20,0xf0,0x09,0x89,0x30,0xa7,0x54
....:    ,0x74,0x8b,0x7d,0xdc,0xb4,0x3e,0xf7,0x5a
....:    ,0x0d,0xbf,0x3a,0x0d,0x26,0x38,0x1a,0xf4
....:    ,0xeb,0xa4,0xa9,0x8e,0xaa,0x9b,0x4e,0x6a]
sage: s == sum(pk[i]*256^i for i in range(32))
True
```

Testing: Python vs. `scalarmult_curve25519_base`. This Python script, contributed by Matthew Dempsky, includes self-contained Curve25519 functions independent of the Sage implementation of elliptic curves:

```
P = 2 ** 255 - 19
A = 486662

def expmod(b, e, m):
  if e == 0: return 1
  t = expmod(b, e / 2, m) ** 2 % m
  if e & 1: t = (t * b) % m
  return t

def inv(x):
  return expmod(x, P - 2, P)

# Addition and doubling formulas taken
# from Appendix D of "Curve25519:
# new Diffie-Hellman speed records".

def add((xn,zn), (xm,zm), (xd,zd)):
  x = 4 * (xm * xn - zm * zn) ** 2 * zd
  z = 4 * (xm * zn - zm * xn) ** 2 * xd
  return (x % P, z % P)

def double((xn,zn)):
  x = (xn ** 2 - zn ** 2) ** 2
  z = 4 * xn * zn * (xn ** 2 + A * xn * zn + zn ** 2)
  return (x % P, z % P)

def curve25519(n, base):
  one = (base,1)
  two = double(one)
  # f(m) evaluates to a tuple
  # containing the mth multiple and the
  # (m+1)th multiple of base.
  def f(m):
    if m == 1: return (one, two)
    (pm, pm1) = f(m / 2)
    if (m & 1):
      return (add(pm, pm1, one), double(pm1))
    return (double(pm), add(pm, pm1, one))
  ((x,z), _) = f(n)
  return (x * inv(z)) % P

def unpack(s):
  if len(s) != 32:
    raise ValueError('Invalid Curve25519 argument')
  return sum(ord(s[i]) << (8 * i) for i in range(32))

def pack(n):
  return ''.join([chr((n >> (8 * i)) & 255) for i in range(32)])

def clamp(n):
  n &= ~7
  n &= ~(128 << 8 * 31)
  n |= 64 << 8 * 31
  return n

def crypto_scalarmult_curve25519(n, p):
  n = clamp(unpack(n))
  p = unpack(p)
  return pack(curve25519(n, p))

def crypto_scalarmult_curve25519_base(n):
  n = clamp(unpack(n))
  return pack(curve25519(n, 9))
```

After this script the extra commands

```
sk=[0x77,0x07,0x6d,0x0a,0x73,0x18,0xa5,0x7d
   ,0x3c,0x16,0xc1,0x72,0x51,0xb2,0x66,0x45
   ,0xdf,0x4c,0x2f,0x87,0xeb,0xc0,0x99,0x2a
   ,0xb1,0x77,0xfb,0xa5,0x1d,0xb9,0x2c,0x2a]
n=''.join([chr(sk[i]) for i in range(32)])
pk=[0x85,0x20,0xf0,0x09,0x89,0x30,0xa7,0x54
   ,0x74,0x8b,0x7d,0xdc,0xb4,0x3e,0xf7,0x5a
   ,0x0d,0xbf,0x3a,0x0d,0x26,0x38,0x1a,0xf4
   ,0xeb,0xa4,0xa9,0x8e,0xaa,0x9b,0x4e,0x6a]
s=''.join([chr(pk[i]) for i in range(32)])
print s == crypto_scalarmult_curve25519_base(n)
```

print `True`.

Chapter 18

Example of the receiver's keys

The following program uses C NaCl to compute the public key corresponding to another secret key:

```
#include <stdio.h>
#include "crypto_scalarmult_curve25519.h"

unsigned char bobsk[32] = {
 0x5d,0xab,0x08,0x7e,0x62,0x4a,0x8a,0x4b
,0x79,0xe1,0x7f,0x8b,0x83,0x80,0x0e,0xe6
,0x6f,0x3b,0xb1,0x29,0x26,0x18,0xb6,0xfd
,0x1c,0x2f,0x8b,0x27,0xff,0x88,0xe0,0xeb
} ;

unsigned char bobpk[32];

main()
{
  int i;
  crypto_scalarmult_curve25519_base(bobpk,bobsk);
  for (i = 0;i < 32;++i) {
    if (i > 0) printf(","); else printf(" ");
    printf("0x%02x",(unsigned int) bobpk[i]);
    if (i % 8 == 7) printf("\n");
  }
  return 0;
}
```

As in the previous chapter, the secret key bytes embedded into the program were copied from output of `od -t x1 /dev/urandom` head -2—. The output of the program is the corresponding public key:

```
 0xde,0x9e,0xdb,0x7d,0x7b,0x7d,0xc1,0xb4
,0xd3,0x5b,0x61,0xc2,0xec,0xe4,0x35,0x37
,0x3f,0x83,0x43,0xc8,0x5b,0x78,0x67,0x4d
,0xad,0xfc,0x7e,0x14,0x6f,0x88,0x2b,0x4f
```

The remaining chapters of Part II will reuse this example, assuming that Bob's keys are the particular secret key and public key shown here.

Testing: Sage vs. `scalarmult_curve25519_base`. A short Sage script clamps Bob's secret key shown above, converts the result to an integer n, computes $n(9, \sqrt{39420360})$ in $E(\mathbf{F}_p)$, and checks that the resulting x-coordinate matches the public key computed by C NaCl:

```
sage: sk=[0x5d,0xab,0x08,0x7e,0x62,0x4a,0x8a,0x4b
....:    ,0x79,0xe1,0x7f,0x8b,0x83,0x80,0x0e,0xe6
....:    ,0x6f,0x3b,0xb1,0x29,0x26,0x18,0xb6,0xfd
....:    ,0x1c,0x2f,0x8b,0x27,0xff,0x88,0xe0,0xeb]
sage: clampsk=sk
sage: clampsk[0]=clampsk[0]-(clampsk[0]%8)
sage: clampsk[31]=64+(clampsk[31]%64)
sage: n=sum(clampsk[i]*256^i for i in range(32))
sage: p=2^255-19
sage: k=GF(p)
sage: E=EllipticCurve([k(0),486662,0,1,0])
sage: s=lift((n*E([k(9),sqrt(k(39420360))]))[0])
sage: pk=[0xde,0x9e,0xdb,0x7d,0x7b,0x7d,0xc1,0xb4
....:    ,0xd3,0x5b,0x61,0xc2,0xec,0xe4,0x35,0x37
....:    ,0x3f,0x83,0x43,0xc8,0x5b,0x78,0x67,0x4d
....:    ,0xad,0xfc,0x7e,0x14,0x6f,0x88,0x2b,0x4f]
sage: s == sum(pk[i]*256^i for i in range(32))
True
```

Testing: Python vs. `scalarmult_curve25519_base`. After the Python script shown in Chapter 17, the extra commands

```
sk=[0x5d,0xab,0x08,0x7e,0x62,0x4a,0x8a,0x4b
   ,0x79,0xe1,0x7f,0x8b,0x83,0x80,0x0e,0xe6
   ,0x6f,0x3b,0xb1,0x29,0x26,0x18,0xb6,0xfd
   ,0x1c,0x2f,0x8b,0x27,0xff,0x88,0xe0,0xeb]
n=''.join([chr(sk[i]) for i in range(32)])
pk=[0xde,0x9e,0xdb,0x7d,0x7b,0x7d,0xc1,0xb4
   ,0xd3,0x5b,0x61,0xc2,0xec,0xe4,0x35,0x37
   ,0x3f,0x83,0x43,0xc8,0x5b,0x78,0x67,0x4d
   ,0xad,0xfc,0x7e,0x14,0x6f,0x88,0x2b,0x4f]
s=''.join([chr(pk[i]) for i in range(32)])
print s == crypto_scalarmult_curve25519_base(n)
```

print `True`.

Chapter 19

Shared secret

At this point Alice has a secret key $a \in \{0, 1, \ldots, 255\}^{32}$ and a public key $A = \text{Curve25519}(\text{ClampC}(a), \underline{9}) \in \{0, 1, \ldots, 255\}^{32}$. Similarly, Bob has a secret key b and a public key $B = \text{Curve25519}(\text{ClampC}(b), \underline{9})$.

Assume that Alice knows Bob's public key from a previous secure channel—for example, from meeting privately with Bob. Similarly assume that Bob knows Alice's public key. There is no hope of security if the previous channel allows forgeries: for example, if an attacker can replace Bob's public key with the attacker's public key then Alice will end up encrypting a packet to the attacker instead of to Bob.

Alice computes $\text{Curve25519}(\text{ClampC}(a), B)$ from her secret key a and Bob's public key B. Bob computes $\text{Curve25519}(\text{ClampC}(b), A)$ from his secret key b and Alice's public key A. The definition of Curve25519 immediately implies that $\text{Curve25519}(\text{ClampC}(a), B) = \text{Curve25519}(\text{ClampC}(b), A)$, so at this point Alice and Bob have computed the same 32-byte string.

In the next step, described in Chapter 21, Alice will convert this 32-byte shared secret k into a 32-byte string $\text{HSalsa20}(k, 0)$, which is then used to encrypt and authenticate packets. Bob similarly uses $\text{HSalsa20}(k, 0)$ to verify and decrypt the packets. No other use is made of k. One can thus view $\text{HSalsa20}(k, 0)$ as the shared secret rather than k.

Security notes beyond ECDLP. An attacker who can solve the elliptic-curve discrete-logarithm problem can figure out Alice's secret key from Alice's public key, and can then compute the shared secret the same way Alice does; or figure out Bob's secret key from Bob's public key, and can then compute the shared secret the same way Bob does.

Computing the shared secret from the two public keys—the "Diffie–Hellman problem"—is widely conjectured to be as difficult as computing discrete logarithms. There are weak theorems along these lines, stating that (for typical elliptic curves) a reliable algorithm to solve the Diffie–Hellman problem can be converted into a discrete-logarithm algorithm costing about ten thousand times as much.

It is much easier to compute *some information* about the 32-byte string k. There are only $p_1 \approx 2^{251}$ possibilities for k, and the set of possibilities for k is an easy-to-recognize set: for example, the last bit of k is always 0. However, $\text{HSalsa20}(k, 0)$ is conjectured to be indistinguishable from $\text{HSalsa20}(k', 0)$ where k' is a uniform random Curve25519 output.

It is often conjectured that the "decision Diffie–Hellman problem" is hard: i.e., that k is indistinguishable from k'. However, this DDH conjecture is overkill. What matters is that $\text{HSalsa20}(k, 0)$ is indistinguishable from $\text{HSalsa20}(k', 0)$.

Alice can reuse her secret key and public key for communicating with many parties. Some of those parties may be attackers with fake public keys—32-byte strings that are not of the form Curve25519(ClampC(...), $\underline{9}$). The corresponding points can be in the "twist group" $E(\mathbf{F}_{p^2}) \cap (\{\infty\} \cup (\mathbf{F}_p \times \sqrt{2}\mathbf{F}_p))$; even if the points are in $E(\mathbf{F}_p)$, they can be outside the subgroup of order p_1. If the points have small order then they can reveal Alice's secret n modulo that order. Fortunately, $E(\mathbf{F}_p)$ has order $8p_1$ by the Hasse–Weil theorem, and the twist group has order $4p_2$ where p_2 is the prime number

$$2^{253} - 55484635554744707071703875581767296995 = (p+1)/2 - 2p_1.$$

The following Sage transcript captures the relevant facts about p_2:

```
sage: p=2^255-19
sage: p1=2^252+27742317777372353535851937790883648493
sage: p2=2^253-55484635554744707071703875581767296995
sage: p2.is_prime()
True
sage: 8*p1+4*p2-2*(p+1)
0
```

Consequently the only possible small orders are 1, 2, 4, and 8, and an attacker can learn at most Alice's $n \bmod 8$, which is always 0 by construction. See [13, Section 3] for further discussion of active attacks and twist security.

Chapter 20

Example of the shared secret

The following program, starting from Chapter 17's example of Alice's secret key and Chapter 18's example of Bob's public key, uses C NaCl to compute the secret shared between Alice and Bob:

```
#include <stdio.h>
#include "crypto_scalarmult_curve25519.h"

unsigned char alicesk[32] = {
 0x77,0x07,0x6d,0x0a,0x73,0x18,0xa5,0x7d
,0x3c,0x16,0xc1,0x72,0x51,0xb2,0x66,0x45
,0xdf,0x4c,0x2f,0x87,0xeb,0xc0,0x99,0x2a
,0xb1,0x77,0xfb,0xa5,0x1d,0xb9,0x2c,0x2a
} ;

unsigned char bobpk[32] = {
 0xde,0x9e,0xdb,0x7d,0x7b,0x7d,0xc1,0xb4
,0xd3,0x5b,0x61,0xc2,0xec,0xe4,0x35,0x37
,0x3f,0x83,0x43,0xc8,0x5b,0x78,0x67,0x4d
,0xad,0xfc,0x7e,0x14,0x6f,0x88,0x2b,0x4f
} ;

unsigned char k[32];

main()
{
  int i;
  crypto_scalarmult_curve25519(k,alicesk,bobpk);
  for (i = 0;i < 32;++i) {
    if (i > 0) printf(","); else printf(" ");
    printf("0x%02x",(unsigned int) k[i]);
    if (i % 8 == 7) printf("\n");
  }
  return 0;
}
```

The program produces the following output:

```
 0x4a,0x5d,0x9d,0x5b,0xa4,0xce,0x2d,0xe1
,0x72,0x8e,0x3b,0xf4,0x80,0x35,0x0f,0x25
,0xe0,0x7e,0x21,0xc9,0x47,0xd1,0x9e,0x33
,0x76,0xf0,0x9b,0x3c,0x1e,0x16,0x17,0x42
```

The following program, starting from Chapter 18's example of Bob's secret key and Chapter 17's example of Alice's public key, uses C NaCl to compute the secret shared between Alice and Bob:

```
#include <stdio.h>
#include "crypto_scalarmult_curve25519.h"

unsigned char bobsk[32] = {
 0x5d,0xab,0x08,0x7e,0x62,0x4a,0x8a,0x4b
,0x79,0xe1,0x7f,0x8b,0x83,0x80,0x0e,0xe6
,0x6f,0x3b,0xb1,0x29,0x26,0x18,0xb6,0xfd
,0x1c,0x2f,0x8b,0x27,0xff,0x88,0xe0,0xeb
} ;

unsigned char alicepk[32] = {
 0x85,0x20,0xf0,0x09,0x89,0x30,0xa7,0x54
,0x74,0x8b,0x7d,0xdc,0xb4,0x3e,0xf7,0x5a
,0x0d,0xbf,0x3a,0x0d,0x26,0x38,0x1a,0xf4
,0xeb,0xa4,0xa9,0x8e,0xaa,0x9b,0x4e,0x6a
} ;

unsigned char k[32];

main()
{
  int i;
  crypto_scalarmult_curve25519(k,bobsk,alicepk);
  for (i = 0;i < 32;++i) {
    if (i > 0) printf(","); else printf(" ");
    printf("0x%02x",(unsigned int) k[i]);
    if (i % 8 == 7) printf("\n");
  }
  return 0;
}
```

This program produces the same output as the previous program.

Testing: Sage vs. `scalarmult_curve25519`**.** A short Sage script clamps Alice's secret key, converts the result to an integer n, clamps Bob's secret key, converts the result to an integer m, computes $mn(9, \sqrt{39420360})$ in $E(\mathbf{F}_p)$, and checks that the x-coordinate of the result matches the shared secret computed by C NaCl:

```
sage: alicesk=[0x77,0x07,0x6d,0x0a,0x73,0x18,0xa5,0x7d
....:         ,0x3c,0x16,0xc1,0x72,0x51,0xb2,0x66,0x45
....:         ,0xdf,0x4c,0x2f,0x87,0xeb,0xc0,0x99,0x2a
....:         ,0xb1,0x77,0xfb,0xa5,0x1d,0xb9,0x2c,0x2a]
sage: clampsk=alicesk
sage: clampsk[0]=clampsk[0]-(clampsk[0]%8)
sage: clampsk[31]=64+(clampsk[31]%64)
sage: n=sum(clampsk[i]*256^i for i in range(32))
sage: bobsk=[0x5d,0xab,0x08,0x7e,0x62,0x4a,0x8a,0x4b
....:       ,0x79,0xe1,0x7f,0x8b,0x83,0x80,0x0e,0xe6
....:       ,0x6f,0x3b,0xb1,0x29,0x26,0x18,0xb6,0xfd
....:       ,0x1c,0x2f,0x8b,0x27,0xff,0x88,0xe0,0xeb]
sage: clampsk=bobsk
sage: clampsk[0]=clampsk[0]-(clampsk[0]%8)
sage: clampsk[31]=64+(clampsk[31]%64)
sage: m=sum(clampsk[i]*256^i for i in range(32))
sage: p=2^255-19
sage: k=GF(p)
sage: E=EllipticCurve([k(0),486662,0,1,0])
sage: s=lift((m*n*E([k(9),sqrt(k(39420360))]))[0])
sage: shared=[0x4a,0x5d,0x9d,0x5b,0xa4,0xce,0x2d,0xe1
....:        ,0x72,0x8e,0x3b,0xf4,0x80,0x35,0x0f,0x25
....:        ,0xe0,0x7e,0x21,0xc9,0x47,0xd1,0x9e,0x33
....:        ,0x76,0xf0,0x9b,0x3c,0x1e,0x16,0x17,0x42]
sage: s == sum(shared[i]*256^i for i in range(32))
True
```

Testing: Python vs. `scalarmult_curve25519`**.** After the Python script shown in Chapter 17, the extra commands

```
alicesk=[0x77,0x07,0x6d,0x0a,0x73,0x18,0xa5,0x7d
        ,0x3c,0x16,0xc1,0x72,0x51,0xb2,0x66,0x45
        ,0xdf,0x4c,0x2f,0x87,0xeb,0xc0,0x99,0x2a
        ,0xb1,0x77,0xfb,0xa5,0x1d,0xb9,0x2c,0x2a]
a=''.join([chr(alicesk[i]) for i in range(32)])
bobpk=[0xde,0x9e,0xdb,0x7d,0x7b,0x7d,0xc1,0xb4
      ,0xd3,0x5b,0x61,0xc2,0xec,0xe4,0x35,0x37
      ,0x3f,0x83,0x43,0xc8,0x5b,0x78,0x67,0x4d
      ,0xad,0xfc,0x7e,0x14,0x6f,0x88,0x2b,0x4f]
b=''.join([chr(bobpk[i]) for i in range(32)])
shared=[0x4a,0x5d,0x9d,0x5b,0xa4,0xce,0x2d,0xe1
       ,0x72,0x8e,0x3b,0xf4,0x80,0x35,0x0f,0x25
       ,0xe0,0x7e,0x21,0xc9,0x47,0xd1,0x9e,0x33
       ,0x76,0xf0,0x9b,0x3c,0x1e,0x16,0x17,0x42]
s=''.join([chr(shared[i]) for i in range(32)])
print s == crypto_scalarmult_curve25519(a,b)
```

print `true`, and the extra commands

```
bobsk=[0x5d,0xab,0x08,0x7e,0x62,0x4a,0x8a,0x4b
      ,0x79,0xe1,0x7f,0x8b,0x83,0x80,0x0e,0xe6
      ,0x6f,0x3b,0xb1,0x29,0x26,0x18,0xb6,0xfd
      ,0x1c,0x2f,0x8b,0x27,0xff,0x88,0xe0,0xeb]
b=''.join([chr(bobsk[i]) for i in range(32)])
alicepk=[0x85,0x20,0xf0,0x09,0x89,0x30,0xa7,0x54
        ,0x74,0x8b,0x7d,0xdc,0xb4,0x3e,0xf7,0x5a
        ,0x0d,0xbf,0x3a,0x0d,0x26,0x38,0x1a,0xf4
        ,0xeb,0xa4,0xa9,0x8e,0xaa,0x9b,0x4e,0x6a]
a=''.join([chr(alicepk[i]) for i in range(32)])
shared=[0x4a,0x5d,0x9d,0x5b,0xa4,0xce,0x2d,0xe1
       ,0x72,0x8e,0x3b,0xf4,0x80,0x35,0x0f,0x25
       ,0xe0,0x7e,0x21,0xc9,0x47,0xd1,0x9e,0x33
       ,0x76,0xf0,0x9b,0x3c,0x1e,0x16,0x17,0x42]
s=''.join([chr(shared[i]) for i in range(32)])
print s == crypto_scalarmult_curve25519(b,a)
```

print `true`.

Chapter 21

Nonce and stream

At this point Alice and Bob have a shared secret $k \in \{0, 1, \ldots, 255\}^{32}$. This secret can be used to protect a practically infinite sequence of packets exchanged between Alice and Bob.

Alice and Bob assign to each packet a **nonce** $n \in \{0, 1, \ldots, 255\}^{24}$: a unique message number that will never be reused for other packets exchanged between Alice and Bob. For example, the nonce can be chosen as a simple counter: 0 for Alice's first packet, 1 for Bob's first packet, 2 for Alice's second packet, 3 for Bob's second packet, 4 for Alice's third packet, 5 for Bob's third packet, etc. Choosing the nonce as a counter followed by (e.g.) 32 random bits helps protect some protocols against denial-of-service attacks. In many applications it is better to increase the counter to, e.g., the number of nanoseconds that have passed since a standard epoch in the local clock, so that the current value of the counter does not leak the traffic rate. Note that "increase" does not mean "increase or decrease"; if the clock jumps backwards, the counter must continue to increase.

Alice uses the shared secret k to expand the nonce n into a long stream. Specifically, Alice computes a first-level key $\text{HSalsa20}(k, 0)$; uses the first 16 bytes n_1 of the nonce to compute a second-level key $\text{HSalsa20}(\text{HSalsa20}(k, 0), n_1)$; and uses the remaining 8 bytes n_2 of the nonce to compute a long stream $\text{Salsa20}(\text{HSalsa20}(\text{HSalsa20}(k, 0), n_1), n_2)$. This stream is then used to encrypt and authenticate the packet, as described in subsequent chapters.

This chapter defines HSalsa20 and Salsa20. Many of the definitions here are copied from the original Salsa20 specification [12]. Chapter 22 gives an example of nonce expansion, starting from the key examples used in Chapters 18, 17, and 20.

Words. A **word** is an element of $\{0, 1, \ldots, 2^{32} - 1\}$.

The **sum** of two words u, v is $u + v \bmod 2^{32}$. The sum is denoted $u + v$; there is no risk of confusion. For example, `0xc0a8787e` + `0x9fd1161d` = `0x60798e9b`.

The **exclusive-or** of two words u, v, denoted $u \oplus v$, is the sum of u and v with carries suppressed. In other words, if $u = \sum_i 2^i u_i$ and $v = \sum 2^i v_i$ then $u \oplus v = \sum_i 2^i (u_i + v_i - 2u_i v_i)$. For example, `0xc0a8787e` ⊕ `0x9fd1161d` = `0x5f796e63`.

For each $c \in \{0, 1, 2, 3, \ldots\}$, the ***c*-bit left rotation** of a word u, denoted $u \lll c$, is the unique nonzero word congruent to $2^c u$ modulo $2^{32} - 1$, except that $0 \lll c = 0$. In other words, if $u = \sum_i 2^i u_i$ then $u \lll c = \sum_i 2^{i+c \bmod 32} u_i$. For example, `0xc0a8787e` $\lll 5$ = `0x150f0fd8`.

The quarterround function. If $y = (y_0, y_1, y_2, y_3) \in \{0, 1, \ldots, 2^{32} - 1\}^4$ then quarterround$(y) \in \{0,1,\ldots,2^{32}-1\}^4$ is defined as (z_0, z_1, z_2, z_3) where

$$\begin{aligned} z_1 &= y_1 \oplus ((y_0 + y_3) \lll 7), \\ z_2 &= y_2 \oplus ((z_1 + y_0) \lll 9), \\ z_3 &= y_3 \oplus ((z_2 + z_1) \lll 13), \\ z_0 &= y_0 \oplus ((z_3 + z_2) \lll 18). \end{aligned}$$

The rowround function. If $y = (y_0, y_1, y_2, y_3, \ldots, y_{15}) \in \{0,1,\ldots,2^{32}-1\}^{16}$ then rowround$(y) \in \{0,1,\ldots,2^{32}-1\}^{16}$ is defined as $(z_0, z_1, z_2, z_3, \ldots, z_{15})$ where

$$\begin{aligned} (z_0, z_1, z_2, z_3) &= \text{quarterround}(y_0, y_1, y_2, y_3), \\ (z_5, z_6, z_7, z_4) &= \text{quarterround}(y_5, y_6, y_7, y_4), \\ (z_{10}, z_{11}, z_8, z_9) &= \text{quarterround}(y_{10}, y_{11}, y_8, y_9), \\ (z_{15}, z_{12}, z_{13}, z_{14}) &= \text{quarterround}(y_{15}, y_{12}, y_{13}, y_{14}). \end{aligned}$$

The columnround function. If $x = (x_0, x_1, \ldots, x_{15}) \in \{0,1,\ldots,2^{32}-1\}^{16}$ then columnround$(x) \in \{0,1,\ldots,2^{32}-1\}^{16}$ is defined as $(y_0, y_1, y_2, y_3, \ldots, y_{15})$ where

$$\begin{aligned} (y_0, y_4, y_8, y_{12}) &= \text{quarterround}(x_0, x_4, x_8, x_{12}), \\ (y_5, y_9, y_{13}, y_1) &= \text{quarterround}(x_5, x_9, x_{13}, x_1), \\ (y_{10}, y_{14}, y_2, y_6) &= \text{quarterround}(x_{10}, x_{14}, x_2, x_6), \\ (y_{15}, y_3, y_7, y_{11}) &= \text{quarterround}(x_{15}, x_3, x_7, x_{11}). \end{aligned}$$

Equivalent formula: $(y_0, y_4, y_8, y_{12}, y_1, y_5, y_9, y_{13}, y_2, y_6, y_{10}, y_{14}, y_3, y_7, y_{11}, y_{15})$ = rowround$(x_0, x_4, x_8, x_{12}, x_1, x_5, x_9, x_{13}, x_2, x_6, x_{10}, x_{14}, x_3, x_7, x_{11}, x_{15})$.

The doubleround function. If $x \in \{0,1,\ldots,2^{32}-1\}^{16}$ then doubleround$(x) \in \{0,1,\ldots,2^{32}-1\}^{16}$ is defined as rowround(columnround(x)).

The littleendian function. If $b = (b_0, b_1, b_2, b_3) \in \{0,1,2,3,\ldots,255\}^4$ then littleendian$(b) \in \{0,1,\ldots,2^{32}-1\}$ is defined as $b_0 + 2^8 b_1 + 2^{16} b_2 + 2^{24} b_3$. More generally, if $b = (b_0, b_1, \ldots, b_{4k-1}) \in \{0,1,\ldots,255\}^{4k}$ then littleendian$(b) \in \{0,1,\ldots,2^{32}-1\}^k$ is defined as

$$(b_0 + 2^8 b_1 + 2^{16} b_2 + 2^{24} b_3, b_4 + 2^8 b_5 + 2^{16} b_6 + 2^{24} b_7, \ldots).$$

Note that littleendian is invertible.

The HSalsa20 function. The function

$$\text{HSalsa20} : \{0,1,\ldots,255\}^{32} \times \{0,1,\ldots,255\}^{16} \to \{0,1,\ldots,255\}^{32}$$

is defined as follows.

Fix $k \in \{0,1,\ldots,255\}^{32}$ and $n \in \{0,1,\ldots,255\}^{16}$. Define $(x_0, x_1, \ldots, x_{15}) \in \{0,1,\ldots,2^{32}-1\}^{16}$ as follows:

- $(x_0, x_5, x_{10}, x_{15})$ = (`0x61707865`, `0x3320646e`, `0x79622d32`, `0x6b206574`); in other words, $(x_0, x_5, x_{10}, x_{15})$ is the **Salsa20 constant**.

- $(x_1, x_2, x_3, x_4, x_{11}, x_{12}, x_{13}, x_{14}) = \text{littleendian}(k)$; and
- $(x_6, x_7, x_8, x_9) = \text{littleendian}(n)$.

Define $(z_0, z_1, \ldots, z_{15}) = \text{doubleround}^{10}(x_0, x_1, \ldots, x_{15})$. Then $\text{HSalsa20}(k, n) = \text{littleendian}^{-1}(z_0, z_5, z_{10}, z_{15}, z_6, z_7, z_8, z_9)$.

The Salsa20 expansion function. The function

$$\text{Salsa20} : \{0, 1, \ldots, 255\}^{32} \times \{0, 1, \ldots, 255\}^{16} \to \{0, 1, \ldots, 255\}^{64}$$

is defined as follows.

Fix $k \in \{0, 1, \ldots, 255\}^{32}$ and $n \in \{0, 1, \ldots, 255\}^{16}$. Define $(x_0, x_1, \ldots, x_{15}) \in \{0, 1, \ldots, 2^{32} - 1\}^{16}$ as follows:

- $(x_0, x_5, x_{10}, x_{15})$ is the Salsa20 constant;
- $(x_1, x_2, x_3, x_4, x_{11}, x_{12}, x_{13}, x_{14}) = \text{littleendian}(k)$; and
- $(x_6, x_7, x_8, x_9) = \text{littleendian}(n)$.

Define $(z_0, z_1, \ldots, z_{15}) = \text{doubleround}^{10}(x_0, x_1, \ldots, x_{15})$. Then $\text{Salsa20}(k, n) = \text{littleendian}^{-1}(x_0 + z_0, x_1 + z_1, \ldots, x_{15} + z_{15})$.

The Salsa20 streaming function. The function

$$\text{Salsa20} : \{0, 1, \ldots, 255\}^{32} \times \{0, 1, \ldots, 255\}^{8} \to \{0, 1, \ldots, 255\}^{2^{70}}$$

is defined as follows: $\text{Salsa20}(k, n) = \text{Salsa20}(k, n, \underline{0}), \text{Salsa20}(k, n, \underline{1}), \ldots$. Here $\underline{b}$ means the 8-byte string $(b \bmod 256, \lfloor b/256 \rfloor \bmod 256, \ldots)$.

Security notes. ECRYPT, a consortium of European research organizations, issued a Call for Stream Cipher Primitives in November 2004, and received 34 submissions from 97 cryptographers in 19 countries. In April 2008, after two hundred papers and several conferences, ECRYPT selected a portfolio of 4 software ciphers and 4 lower-security hardware ciphers.

Bernstein submitted Salsa20. Later he suggested the reduced-round variants Salsa20/12 and Salsa20/8 (replacing doubleround^{10} with doubleround^{6} and doubleround^{4} respectively) as higher-speed options for users who value speed more highly than confidence. Four attack papers by fourteen cryptanalysts ([20], [23], [33], and [4]) culminated in a 2^{184}-operation attack on Salsa20/7 and a 2^{251}-operation attack on Salsa20/8. The eSTREAM portfolio recommended Salsa20/12: "Eight and twenty round versions were also considered during the eSTREAM process, but we feel that Salsa20/12 offers the best balance, combining a very nice performance profile with what appears to be a comfortable margin for security."

The standard ("PRF") security conjecture for Salsa20 is that the Salsa20 output blocks, for a uniform random secret key k, are indistinguishable from independent uniform random 64-byte strings. This conjecture implies the analogous security conjecture for HSalsa20: by [16, Theorem 3.3], any attack against HSalsa20 can be converted into an attack against Salsa20 having exactly the same effectiveness and essentially the same speed.

This conjecture also implies an analogous security conjecture for the cascade $(n_1, n_2) \mapsto \text{Salsa20}(\text{HSalsa20}(\text{HSalsa20}(k, 0), n_1), n_2)$: by [16, Theorem 3.1], any q-query attack against the cascade can be converted into an attack against Salsa20 having at least $1/(2q + 1)$ as much effectiveness and essentially the same speed.

A Curve25519 output k is not a uniform random 32-byte string, but any attack against a uniform random Curve25519 output can be converted into an attack against a uniform random 32-byte string having at least $1/32$ as much effectiveness and essentially the same speed—and therefore an attack against Salsa20 having at least $1/(64q + 32)$ as much effectiveness and essentially the same speed.

Chapter 22

Example of the long stream

The following program starts from Chapter 20's example of a shared secret k and uses C NaCl to compute the first-level key $k_1 = \text{HSalsa20}(k, 0)$:

```
#include <stdio.h>
#include "crypto_core_hsalsa20.h"

unsigned char shared[32] = {
 0x4a,0x5d,0x9d,0x5b,0xa4,0xce,0x2d,0xe1
,0x72,0x8e,0x3b,0xf4,0x80,0x35,0x0f,0x25
,0xe0,0x7e,0x21,0xc9,0x47,0xd1,0x9e,0x33
,0x76,0xf0,0x9b,0x3c,0x1e,0x16,0x17,0x42
} ;

unsigned char zero[32] = { 0 };

unsigned char c[16] = {
 0x65,0x78,0x70,0x61,0x6e,0x64,0x20,0x33
,0x32,0x2d,0x62,0x79,0x74,0x65,0x20,0x6b
} ;

unsigned char firstkey[32];

main()
{
  int i;
  crypto_core_hsalsa20(firstkey,zero,shared,c);
  for (i = 0;i < 32;++i) {
    if (i > 0) printf(","); else printf(" ");
    printf("0x%02x",(unsigned int) firstkey[i]);
    if (i % 8 == 7) printf("\n");
  }
  return 0;
}
```

The program prints the following output:

```
 0x1b,0x27,0x55,0x64,0x73,0xe9,0x85,0xd4
,0x62,0xcd,0x51,0x19,0x7a,0x9a,0x46,0xc7
,0x60,0x09,0x54,0x9e,0xac,0x64,0x74,0xf2
,0x06,0xc4,0xee,0x08,0x44,0xf6,0x83,0x89
```

The following program starts from this k_1 example and a sample nonce prefix n_1, and uses C NaCl to compute the second-level key $k_2 = \text{HSalsa20}(k_1, n_1)$:

```
#include <stdio.h>
#include "crypto_core_hsalsa20.h"

unsigned char firstkey[32] = {
 0x1b,0x27,0x55,0x64,0x73,0xe9,0x85,0xd4
,0x62,0xcd,0x51,0x19,0x7a,0x9a,0x46,0xc7
,0x60,0x09,0x54,0x9e,0xac,0x64,0x74,0xf2
,0x06,0xc4,0xee,0x08,0x44,0xf6,0x83,0x89
} ;

unsigned char nonceprefix[16] = {
 0x69,0x69,0x6e,0xe9,0x55,0xb6,0x2b,0x73
,0xcd,0x62,0xbd,0xa8,0x75,0xfc,0x73,0xd6
} ;

unsigned char c[16] = {
 0x65,0x78,0x70,0x61,0x6e,0x64,0x20,0x33
,0x32,0x2d,0x62,0x79,0x74,0x65,0x20,0x6b
} ;

unsigned char secondkey[32];

main()
{
  int i;
  crypto_core_hsalsa20(secondkey,nonceprefix,firstkey,c);
  for (i = 0;i < 32;++i) {
    if (i > 0) printf(","); else printf(" ");
    printf("0x%02x",(unsigned int) secondkey[i]);
    if (i % 8 == 7) printf("\n");
  }
  return 0;
}
```

The program prints the following output:

```
 0xdc,0x90,0x8d,0xda,0x0b,0x93,0x44,0xa9
,0x53,0x62,0x9b,0x73,0x38,0x20,0x77,0x88
,0x80,0xf3,0xce,0xb4,0x21,0xbb,0x61,0xb9
,0x1c,0xbd,0x4c,0x3e,0x66,0x25,0x6c,0xe4
```

The following program starts from this k_2 example and an example of a nonce suffix n_2, and uses C NaCl to print (in binary format) the first 4194304 bytes of Salsa20(k_2, n_2):

```
#include <stdio.h>
#include "crypto_core_salsa20.h"

unsigned char secondkey[32] = {
 0xdc,0x90,0x8d,0xda,0x0b,0x93,0x44,0xa9
,0x53,0x62,0x9b,0x73,0x38,0x20,0x77,0x88
,0x80,0xf3,0xce,0xb4,0x21,0xbb,0x61,0xb9
,0x1c,0xbd,0x4c,0x3e,0x66,0x25,0x6c,0xe4
} ;

unsigned char noncesuffix[8] = {
 0x82,0x19,0xe0,0x03,0x6b,0x7a,0x0b,0x37
} ;

unsigned char c[16] = {
 0x65,0x78,0x70,0x61,0x6e,0x64,0x20,0x33
,0x32,0x2d,0x62,0x79,0x74,0x65,0x20,0x6b
} ;

unsigned char in[16] = { 0 } ;

unsigned char outputblock[64];

main()
{
  int i;
  for (i = 0;i < 8;++i) in[i] = noncesuffix[i];
  do {
    do {
      crypto_core_salsa20(outputblock,in,secondkey,c);
      for (i = 0;i < 64;++i) putchar(outputblock[i]);
    } while (++in[8]);
  } while (++in[9]);
  return 0;
}
```

`662b9d0e3463029156069b12f918691a98f7dfb2ca0393c96bbfc6b1fbd630a2` is the SHA-256 checksum of the output.

Testing: `core_salsa20` **vs.** `stream_salsa20`**.** The following program has the same output as the previous program, but uses `crypto_stream_salsa20` to generate the entire output stream at once:

```
#include <stdio.h>
#include "crypto_stream_salsa20.h"
```

```
unsigned char secondkey[32] = {
 0xdc,0x90,0x8d,0xda,0x0b,0x93,0x44,0xa9
,0x53,0x62,0x9b,0x73,0x38,0x20,0x77,0x88
,0x80,0xf3,0xce,0xb4,0x21,0xbb,0x61,0xb9
,0x1c,0xbd,0x4c,0x3e,0x66,0x25,0x6c,0xe4
} ;

unsigned char noncesuffix[8] = {
 0x82,0x19,0xe0,0x03,0x6b,0x7a,0x0b,0x37
} ;

unsigned char output[4194304];

main()
{
  int i;
  crypto_stream_salsa20(output,4194304,noncesuffix,secondkey);
  for (i = 0;i < 4194304;++i) putchar(output[i]);
  return 0;
}
```

Testing: `core_salsa20` **vs.** `stream_xsalsa20`**.** The following program has the same output as the previous two programs, but uses `crypto_stream_xsalsa20` to generate the entire output stream starting from the first-level key k_1 and the complete nonce $n = (n_1, n_2)$:

```
#include <stdio.h>
#include "crypto_stream_xsalsa20.h"

unsigned char firstkey[32] = {
 0x1b,0x27,0x55,0x64,0x73,0xe9,0x85,0xd4
,0x62,0xcd,0x51,0x19,0x7a,0x9a,0x46,0xc7
,0x60,0x09,0x54,0x9e,0xac,0x64,0x74,0xf2
,0x06,0xc4,0xee,0x08,0x44,0xf6,0x83,0x89
} ;

unsigned char nonce[24] = {
 0x69,0x69,0x6e,0xe9,0x55,0xb6,0x2b,0x73
,0xcd,0x62,0xbd,0xa8,0x75,0xfc,0x73,0xd6
,0x82,0x19,0xe0,0x03,0x6b,0x7a,0x0b,0x37
} ;

unsigned char output[4194304];

main()
{
  int i;
  crypto_stream_xsalsa20(output,4194304,nonce,firstkey);
  for (i = 0;i < 4194304;++i) putchar(output[i]);
  return 0;
}
```

Testing: Python vs. `core_hsalsa20`. The following Python script, based in part on a script contributed by Matthew Dempsky, computes HSalsa20$(k, 0)$ and compares the result to the k_1 computed by C NaCl:

```
import struct

def rotate(x, n):
  x &= 0xffffffff
  return ((x << n) | (x >> (32 - n))) & 0xffffffff

def step(s, i, j, k, r):
  s[i] ^= rotate(s[j] + s[k],r)

def quarterround(s, i0, i1, i2, i3):
  step(s, i1, i0, i3, 7)
  step(s, i2, i1, i0, 9)
  step(s, i3, i2, i1, 13)
  step(s, i0, i3, i2, 18)

def rowround(s):
  quarterround(s, 0, 1, 2, 3)
  quarterround(s, 5, 6, 7, 4)
  quarterround(s, 10, 11, 8, 9)
  quarterround(s, 15, 12, 13, 14)

def columnround(s):
  quarterround(s, 0, 4, 8, 12)
  quarterround(s, 5, 9, 13, 1)
  quarterround(s, 10, 14, 2, 6)
  quarterround(s, 15, 3, 7, 11)

def doubleround(s):
  columnround(s)
  rowround(s)

def hsalsa20(n,k):
  n=''.join([chr(n[i]) for i in range(16)])
  n = struct.unpack('<4I', n)
  k=''.join([chr(k[i]) for i in range(32)])
  k = struct.unpack('<8I', k)
  s = [0] * 16
  s[::5] = struct.unpack('<4I', 'expand 32-byte k')
  s[1:5] = k[:4]
  s[6:10] = n
  s[11:15] = k[4:]
  for i in range(10): doubleround(s)
  s = [s[i] for i in [0,5,10,15,6,7,8,9]]
  return struct.pack('<8I',*s)

k = [0x4a,0x5d,0x9d,0x5b,0xa4,0xce,0x2d,0xe1
    ,0x72,0x8e,0x3b,0xf4,0x80,0x35,0x0f,0x25
    ,0xe0,0x7e,0x21,0xc9,0x47,0xd1,0x9e,0x33
    ,0x76,0xf0,0x9b,0x3c,0x1e,0x16,0x17,0x42]
n = [0] * 16

expected=[0x1b,0x27,0x55,0x64,0x73,0xe9,0x85,0xd4
         ,0x62,0xcd,0x51,0x19,0x7a,0x9a,0x46,0xc7
         ,0x60,0x09,0x54,0x9e,0xac,0x64,0x74,0xf2
         ,0x06,0xc4,0xee,0x08,0x44,0xf6,0x83,0x89]
expected = ''.join([chr(expected[i]) for i in range(32)])

print hsalsa20(n,k) == expected
```

The script prints `True`.

The following extra commands compute HSalsa20(k_1, n_1), where n_1 is the nonce prefix shown above, and compare the result to the k_2 computed by C NaCl:

```
k=[0x1b,0x27,0x55,0x64,0x73,0xe9,0x85,0xd4
  ,0x62,0xcd,0x51,0x19,0x7a,0x9a,0x46,0xc7
  ,0x60,0x09,0x54,0x9e,0xac,0x64,0x74,0xf2
  ,0x06,0xc4,0xee,0x08,0x44,0xf6,0x83,0x89]
n=[0x69,0x69,0x6e,0xe9,0x55,0xb6,0x2b,0x73
  ,0xcd,0x62,0xbd,0xa8,0x75,0xfc,0x73,0xd6]

expected = [0xdc,0x90,0x8d,0xda,0x0b,0x93,0x44,0xa9
           ,0x53,0x62,0x9b,0x73,0x38,0x20,0x77,0x88
           ,0x80,0xf3,0xce,0xb4,0x21,0xbb,0x61,0xb9
           ,0x1c,0xbd,0x4c,0x3e,0x66,0x25,0x6c,0xe4]
expected = ''.join([chr(expected[i]) for i in range(32)])

print hsalsa20(n,k) == expected
```

These commands print `True`.

Testing: Python vs. `stream_salsa20`. The following Python script, based in part on a script contributed by Matthew Dempsky, computes the first 4194304 bytes of Salsa20(k_2, n_2), for the sample k_2, n_2 shown above:

```
import struct
import sys

def rotate(x, n):
  x &= 0xffffffff
  return ((x << n) | (x >> (32 - n))) & 0xffffffff

def step(s, i, j, k, r):
  s[i] ^= rotate(s[j] + s[k],r)

def quarterround(s, i0, i1, i2, i3):
  step(s, i1, i0, i3, 7)
  step(s, i2, i1, i0, 9)
  step(s, i3, i2, i1, 13)
  step(s, i0, i3, i2, 18)

def rowround(s):
  quarterround(s, 0, 1, 2, 3)
  quarterround(s, 5, 6, 7, 4)
  quarterround(s, 10, 11, 8, 9)
  quarterround(s, 15, 12, 13, 14)

def columnround(s):
  quarterround(s, 0, 4, 8, 12)
  quarterround(s, 5, 9, 13, 1)
  quarterround(s, 10, 14, 2, 6)
  quarterround(s, 15, 3, 7, 11)

def doubleround(s):
  columnround(s)
  rowround(s)

def rounds(s, n):
  s1 = list(s)
  while n >= 2:
    doubleround(s1)
    n -= 2
  for i in range(16): s[i] = (s[i] + s1[i]) & 0xffffffff

o = struct.unpack('<4I', 'expand 32-byte k')

def block(i, n, k):
  i = i / 64
  i = (i & 0xffffffff, i >> 32)
  s = [0] * 16
  s[::5] = o
  s[1:5] = k[:4]
  s[6:10] = n + i
  s[11:15] = k[4:]
  rounds(s, 20)
  return struct.pack('<16I', *s)
```

```
def print_salsa20(l, n, k):
  n = struct.unpack('<2I', n)
  k = struct.unpack('<8I', k)
  for i in xrange(0, l, 64):
    sys.stdout.write(block(i, n, k)[:l-i])

k=[0xdc,0x90,0x8d,0xda,0x0b,0x93,0x44,0xa9
  ,0x53,0x62,0x9b,0x73,0x38,0x20,0x77,0x88
  ,0x80,0xf3,0xce,0xb4,0x21,0xbb,0x61,0xb9
  ,0x1c,0xbd,0x4c,0x3e,0x66,0x25,0x6c,0xe4]
k = ''.join([chr(k[i]) for i in range(32)])

n=[0x82,0x19,0xe0,0x03,0x6b,0x7a,0x0b,0x37]
n = ''.join([chr(n[i]) for i in range(8)])

print_salsa20(4194304,n,k)
```

The output is the same as the 4194304-byte output from the C NaCl program shown earlier.

Testing: Salsa20 specification vs. `core_salsa20`. The following program uses C NaCl to compute the first Salsa20 example in [12, Section 9]:

```
#include <stdio.h>
#include "crypto_core_salsa20.h"

unsigned char k[32] = {
   1,  2,  3,  4,  5,  6,  7,  8
,  9, 10, 11, 12, 13, 14, 15, 16
,201,202,203,204,205,206,207,208
,209,210,211,212,213,214,215,216
} ;

unsigned char in[16] = {
 101,102,103,104,105,106,107,108
,109,110,111,112,113,114,115,116
} ;

unsigned char c[16] = {
 101,120,112, 97,110,100, 32, 51
, 50, 45, 98,121,116,101, 32,107
} ;

unsigned char out[64];

main()
{
  int i;
  crypto_core_salsa20(out,in,k,c);
  for (i = 0;i < 64;++i) {
    if (i > 0) printf(","); else printf(" ");
    printf("%3d",(unsigned int) out[i]);
    if (i % 8 == 7) printf("\n");
  }
  return 0;
}
```

The output of the program is

```
  69, 37, 68, 39, 41, 15,107,193
,255,139,122,  6,170,233,217, 98
, 89,144,182,106, 21, 51,200, 65
,239, 49,222, 34,215,114, 40,126
,104,197,  7,225,197,153, 31,  2
,102, 78, 76,176, 84,245,246,184
,177,160,133,130,  6, 72,149,119
,192,195,132,236,234,103,246, 74
```

matching the output shown in [12, Section 9].

Testing: `core_salsa20` **vs.** `core_hsalsa20`**.** The following program uses C NaCl to compute HSalsa20 on a sample input:

```
#include <stdio.h>
#include "crypto_core_hsalsa20.h"

unsigned char k[32] = {
 0xee,0x30,0x4f,0xca,0x27,0x00,0x8d,0x8c
,0x12,0x6f,0x90,0x02,0x79,0x01,0xd8,0x0f
,0x7f,0x1d,0x8b,0x8d,0xc9,0x36,0xcf,0x3b
,0x9f,0x81,0x96,0x92,0x82,0x7e,0x57,0x77
} ;

unsigned char in[16] = {
 0x81,0x91,0x8e,0xf2,0xa5,0xe0,0xda,0x9b
,0x3e,0x90,0x60,0x52,0x1e,0x4b,0xb3,0x52
} ;

unsigned char c[16] = {
 101,120,112, 97,110,100, 32, 51
, 50, 45, 98,121,116,101, 32,107
} ;

unsigned char out[32];

main()
{
  int i;
  crypto_core_hsalsa20(out,in,k,c);
  for (i = 0;i < 32;++i) {
    printf(",0x%02x",(unsigned int) out[i]);
    if (i % 8 == 7) printf("\n");
  }
  return 0;
}
```

Here is the output of the program:

```
,0xbc,0x1b,0x30,0xfc,0x07,0x2c,0xc1,0x40
,0x75,0xe4,0xba,0xa7,0x31,0xb5,0xa8,0x45
,0xea,0x9b,0x11,0xe9,0xa5,0x19,0x1f,0x94
,0xe1,0x8c,0xba,0x8f,0xd8,0x21,0xa7,0xcd
```

The following program uses C NaCl to compute Salsa20 on the same sample input, and then converts the Salsa20 output to HSalsa20 output:

```
#include <stdio.h>
#include "crypto_core_salsa20.h"

unsigned char k[32] = {
 0xee,0x30,0x4f,0xca,0x27,0x00,0x8d,0x8c
,0x12,0x6f,0x90,0x02,0x79,0x01,0xd8,0x0f
,0x7f,0x1d,0x8b,0x8d,0xc9,0x36,0xcf,0x3b
,0x9f,0x81,0x96,0x92,0x82,0x7e,0x57,0x77
} ;

unsigned char in[16] = {
 0x81,0x91,0x8e,0xf2,0xa5,0xe0,0xda,0x9b
,0x3e,0x90,0x60,0x52,0x1e,0x4b,0xb3,0x52
} ;

unsigned char c[16] = {
 101,120,112, 97,110,100, 32, 51
, 50, 45, 98,121,116,101, 32,107
} ;

unsigned char out[64];

void print(unsigned char *x,unsigned char *y)
{
  int i;
  unsigned int borrow = 0;
  for (i = 0;i < 4;++i) {
    unsigned int xi = x[i];
    unsigned int yi = y[i];
    printf(",0x%02x",255 & (xi - yi - borrow));
    borrow = (xi < yi + borrow);
  }
}

main()
{
  crypto_core_salsa20(out,in,k,c);
  print(out,c);
  print(out + 20,c + 4); printf("\n");
  print(out + 40,c + 8);
  print(out + 60,c + 12); printf("\n");
  print(out + 24,in);
  print(out + 28,in + 4); printf("\n");
  print(out + 32,in + 8);
  print(out + 36,in + 12); printf("\n");
  return 0;
}
```

This program produces the same output as the previous program.

Chapter 23

Plaintext, ciphertext, and authenticator

To encrypt a packet $m \in \{0,1,\ldots,255\}^{\{0,1,\ldots,2^{70}-32\}}$ using the packet's nonce $n \in \{0,1,\ldots,255\}^{24}$ and the shared secret $k \in \{0,1,\ldots,255\}^{32}$, Alice xors the packet with *part of* the long stream computed in the previous chapter. Alice then uses a different part of the long stream to authenticate the ciphertext. Alice's boxed packet is the authenticator followed by the ciphertext.

Specifically, write the nonce n as (n_1, n_2) with $n_1 \in \{0,1,\ldots,255\}^{16}$ and $n_2 \in \{0,1,\ldots,255\}^8$, and write $\text{Salsa20}(\text{HSalsa20}(\text{HSalsa20}(k,0),n_1),n_2)$ as $(r,s,t,\ldots)$ where $r,s \in \{0,1,\ldots,255\}^{16}$ and $\operatorname{len} t = \operatorname{len} m$. Define $c = m \oplus t \in \{0,1,\ldots,255\}^{\operatorname{len} m}$ and $a = \text{Poly1305}(\text{ClampP}(r), c, s) \in \{0,1,\ldots,255\}^{16}$. The boxed packet is then $(a,c) \in \{0,1,\ldots,255\}^{16+\operatorname{len} m}$.

This chapter defines Poly1305 and ClampP. Some of the definitions here are copied from the original Poly1305 specification [11]; $\text{Poly1305}(r,c,s)$ here is $\text{Poly1305}_r(c,s)$ in the notation of [11].

The ClampP function. The function

$$\text{ClampP} : \{0,1,\ldots,255\}^{16} \to \{0,1,\ldots,255\}^{16}$$

maps $(r_0, r_1, \ldots, r_{15})$ to

$$\begin{array}{llll}(r_0, & r_1, & r_2, & r_3 \bmod 16, \\ r_4 - (r_4 \bmod 4), & r_5, & r_6, & r_7 \bmod 16, \\ r_8 - (r_8 \bmod 4), & r_9, & r_{10}, & r_{11} \bmod 16, \\ r_{12} - (r_{12} \bmod 4), & r_{13}, & r_{14}, & r_{15} \bmod 16).\end{array}$$

The Poly1305 function. Fix $\ell \in \{0,1,\ldots,2^{70}-32\}$, fix $c \in \{0,1,\ldots,255\}^{\ell}$, fix $R \in \{0,1,\ldots,2^{128}-1\}$, and fix $S \in \{0,1,\ldots,2^{128}-1\}$. Write $q = \lceil \ell/16 \rceil$. Write c as $(c[0], c[1], \ldots, c[\ell-1])$. Define $C_1, C_2, \ldots, C_q \in \{1,2,3,\ldots,2^{129}\}$ as follows: if $1 \le i \le \lfloor \ell/16 \rfloor$ then

$$C_i = c[16i-16] + 2^8 c[16i-15] + 2^{16} c[16i-14] + \cdots + 2^{120} c[16i-1] + 2^{128};$$

if ℓ is not a multiple of 16 then

$$C_q = c[16q-16] + 2^8 c[16q-15] + \cdots + 2^{8(\ell \bmod 16)-8} c[\ell-1] + 2^{8(\ell \bmod 16)}.$$

In other words: Pad each 16-byte chunk of the ciphertext to 17 bytes by appending a 1. If the ciphertext has a final chunk between 1 and 15 bytes, append 1 to the chunk, and then zero-pad the chunk to 17 bytes. Either way, treat the resulting 17-byte chunk as an unsigned little-endian integer.

Now Poly1305($\underline{R}, c, \underline{S}$) = $\underline{A}$ where

$$A = (((C_1 R^q + C_2 R^{q-1} + \cdots + C_q R^1) \bmod 2^{130} - 5) + S) \bmod 2^{128}.$$

Here $\underline{A}$ means the 16-byte string ($A \bmod 256, \lfloor A/256 \rfloor \bmod 256, \ldots$); $\underline{R}$ and $\underline{S}$ are defined in the same way.

Security notes. The constructions in this chapter—xor for encryption and Poly1305 for authentication—are *provably* secure. If the attacker cannot distinguish the stream (r, s, t) from a uniform random string then the attacker learns nothing about the original packet m, aside from its length, and has negligible chance of replacing the boxed packet (a, c) with a different packet (a', c') that satisfies $a' = \text{Poly1305}(\text{ClampP}(r), c', s)$. Of course, this guarantee says nothing about an attacker who *can* distinguish (r, s, t) from a uniform random string—for example, an attacker who uses a quantum computer to break elliptic-curve cryptography has as much power as Alice and Bob.

A security proof for Poly1305 appears in [11]. The proof shows that if packets are limited to L bytes then the attacker's success chance for a forgery attempt (a', c') is at most $8\lceil L/16 \rceil / 2^{106}$. Here are some of the critical points in the proof: $2^{130} - 5$ is prime; ClampP(r) is uniformly distributed among 2^{106} possibilities; and distinct strings c produce distinct polynomials $C_1 x^q + C_2 x^{q-1} + \cdots + C_q x^1$ modulo $2^{130} - 5$.

What happens if an attacker is astonishingly lucky and succeeds at a forgery attempt? Presumably this success will be visible from the receiver's behavior. The attacker can then, by polynomial root-finding, easily determine ClampP(r) and s, or at worst a short list of possibilities for ClampP(r) and s, allowing the attacker to generate "re-forgeries" (a'', c'') under the same nonce. However, if the receiver follows the standard practice of insisting on a strictly increasing sequence of nonces, then the receiver will reject all of these "re-forgeries," as pointed out in 2005 by Nyberg, Gilbert, and Robshaw and independently in 2006 by Lange. See [30] and [15, Section 2.5].

If r were reused from one nonce to another, with s generated anew for each nonce, then the first forgery would still be difficult (as pointed out by Wegman and Carter in [37, Section 4]), but after seeing a successful forgery the attacker would be able to generate "re-forgeries" under *other* nonces. If >100-bit security were scaled down to much lower security then the attacker could reasonably hope for this situation to occur. Many authentication systems in the literature have this problem. The following comment appears in [10, Section 8] and was already online in 2000:

> Some writers claim that forgery probabilities around $1/2^{32}$ are adequate for most applications. The attacker's cost of 2^{32} forgery attempts, they say, is much larger than the attacker's benefit from forging a single message. Unfortunately, even if all attackers acted on the basis of rational economic analyses, this argument would be wrong, because it wildly underestimates the attacker's benefit. *In a typical*

> *authentication system, as soon as the attacker is lucky enough to succeed at a few forgeries, he can immediately figure out enough secret information to let him forge messages of his choice.* (This does not contradict the information-theoretic security expressed by Theorem 8.2; the attacker is gaining information from the receiver, not from the sender.) It is crucial for the forgery probability to be so small that attackers have no hope.

(Emphasis added.) Detailed explanations of various re-forgery attacks appeared in [27], [25], and [19].

Attacks of that type do not apply to Poly1305 as used in NaCl. There is a new Poly1305 key (r, s) for each nonce; the standard security conjecture for Salsa20 implies that the keys (r, s) for different nonces are indistinguishable from independent uniform random keys. More importantly, the $>$100-bit security level of Poly1305 prevents forgery attempts from succeeding in the first place.

Chapter 24

Example of the plaintext, ciphertext, and authenticator

The following program starts from Chapter 17's example of Alice's secret key a Chapter 18's example of Bob's public key B, Chapter 22's example of a nonce n, and a sample 131-byte packet, and uses C NaCl to compute the corresponding boxed packet:

```
#include <stdio.h>
#include "crypto_box_curve25519xsalsa20poly1305.h"

unsigned char alicesk[32] = {
 0x77,0x07,0x6d,0x0a,0x73,0x18,0xa5,0x7d
,0x3c,0x16,0xc1,0x72,0x51,0xb2,0x66,0x45
,0xdf,0x4c,0x2f,0x87,0xeb,0xc0,0x99,0x2a
,0xb1,0x77,0xfb,0xa5,0x1d,0xb9,0x2c,0x2a
} ;

unsigned char bobpk[32] = {
 0xde,0x9e,0xdb,0x7d,0x7b,0x7d,0xc1,0xb4
,0xd3,0x5b,0x61,0xc2,0xec,0xe4,0x35,0x37
,0x3f,0x83,0x43,0xc8,0x5b,0x78,0x67,0x4d
,0xad,0xfc,0x7e,0x14,0x6f,0x88,0x2b,0x4f
} ;

unsigned char nonce[24] = {
 0x69,0x69,0x6e,0xe9,0x55,0xb6,0x2b,0x73
,0xcd,0x62,0xbd,0xa8,0x75,0xfc,0x73,0xd6
,0x82,0x19,0xe0,0x03,0x6b,0x7a,0x0b,0x37
} ;

// API requires first 32 bytes to be 0
unsigned char m[163] = {
    0,   0,   0,   0,   0,   0,   0,   0
,   0,   0,   0,   0,   0,   0,   0,   0
,   0,   0,   0,   0,   0,   0,   0,   0
,   0,   0,   0,   0,   0,   0,   0,   0
,0xbe,0x07,0x5f,0xc5,0x3c,0x81,0xf2,0xd5
,0xcf,0x14,0x13,0x16,0xeb,0xeb,0x0c,0x7b
,0x52,0x28,0xc5,0x2a,0x4c,0x62,0xcb,0xd4
,0x4b,0x66,0x84,0x9b,0x64,0x24,0x4f,0xfc
,0xe5,0xec,0xba,0xaf,0x33,0xbd,0x75,0x1a
,0x1a,0xc7,0x28,0xd4,0x5e,0x6c,0x61,0x29
,0x6c,0xdc,0x3c,0x01,0x23,0x35,0x61,0xf4
,0x1d,0xb6,0x6c,0xce,0x31,0x4a,0xdb,0x31
,0x0e,0x3b,0xe8,0x25,0x0c,0x46,0xf0,0x6d
,0xce,0xea,0x3a,0x7f,0xa1,0x34,0x80,0x57
,0xe2,0xf6,0x55,0x6a,0xd6,0xb1,0x31,0x8a
,0x02,0x4a,0x83,0x8f,0x21,0xaf,0x1f,0xde
,0x04,0x89,0x77,0xeb,0x48,0xf5,0x9f,0xfd
,0x49,0x24,0xca,0x1c,0x60,0x90,0x2e,0x52
,0xf0,0xa0,0x89,0xbc,0x76,0x89,0x70,0x40
,0xe0,0x82,0xf9,0x37,0x76,0x38,0x48,0x64
,0x5e,0x07,0x05
} ;

unsigned char c[163];

main()
{
  int i;
  crypto_box_curve25519xsalsa20poly1305(
    c,m,163,nonce,bobpk,alicesk
  );
  for (i = 16;i < 163;++i) {
    printf(",0x%02x",(unsigned int) c[i]);
    if (i % 8 == 7) printf("\n");
  }
  printf("\n");
  return 0;
}
```

The program prints a 147-byte boxed packet:

```
,0xf3,0xff,0xc7,0x70,0x3f,0x94,0x00,0xe5
,0x2a,0x7d,0xfb,0x4b,0x3d,0x33,0x05,0xd9
,0x8e,0x99,0x3b,0x9f,0x48,0x68,0x12,0x73
,0xc2,0x96,0x50,0xba,0x32,0xfc,0x76,0xce
,0x48,0x33,0x2e,0xa7,0x16,0x4d,0x96,0xa4
,0x47,0x6f,0xb8,0xc5,0x31,0xa1,0x18,0x6a
,0xc0,0xdf,0xc1,0x7c,0x98,0xdc,0xe8,0x7b
,0x4d,0xa7,0xf0,0x11,0xec,0x48,0xc9,0x72
,0x71,0xd2,0xc2,0x0f,0x9b,0x92,0x8f,0xe2
,0x27,0x0d,0x6f,0xb8,0x63,0xd5,0x17,0x38
```

```
,0xb4,0x8e,0xee,0xe3,0x14,0xa7,0xcc,0x8a
,0xb9,0x32,0x16,0x45,0x48,0xe5,0x26,0xae
,0x90,0x22,0x43,0x68,0x51,0x7a,0xcf,0xea
,0xbd,0x6b,0xb3,0x73,0x2b,0xc0,0xe9,0xda
,0x99,0x83,0x2b,0x61,0xca,0x01,0xb6,0xde
,0x56,0x24,0x4a,0x9e,0x88,0xd5,0xf9,0xb3
,0x79,0x73,0xf6,0x22,0xa4,0x3d,0x14,0xa6
,0x59,0x9b,0x1f,0x65,0x4c,0xb4,0x5a,0x74
,0xe3,0x55,0xa5
```

The following program starts from Chapter 18's example of Bob's secret key b, Chapter 17's example of Alice's public key A, Chapter 22's example of a nonce n, and the 147-byte boxed packet shown above, and uses C NaCl to open the box:

```
#include <stdio.h>
#include "crypto_box_curve25519xsalsa20poly1305.h"

unsigned char bobsk[32] = {
 0x5d,0xab,0x08,0x7e,0x62,0x4a,0x8a,0x4b
,0x79,0xe1,0x7f,0x8b,0x83,0x80,0x0e,0xe6
,0x6f,0x3b,0xb1,0x29,0x26,0x18,0xb6,0xfd
,0x1c,0x2f,0x8b,0x27,0xff,0x88,0xe0,0xeb
} ;

unsigned char alicepk[32] = {
 0x85,0x20,0xf0,0x09,0x89,0x30,0xa7,0x54
,0x74,0x8b,0x7d,0xdc,0xb4,0x3e,0xf7,0x5a
,0x0d,0xbf,0x3a,0x0d,0x26,0x38,0x1a,0xf4
,0xeb,0xa4,0xa9,0x8e,0xaa,0x9b,0x4e,0x6a
} ;

unsigned char nonce[24] = {
 0x69,0x69,0x6e,0xe9,0x55,0xb6,0x2b,0x73
,0xcd,0x62,0xbd,0xa8,0x75,0xfc,0x73,0xd6
,0x82,0x19,0xe0,0x03,0x6b,0x7a,0x0b,0x37
} ;

// API requires first 16 bytes to be 0
unsigned char c[163] = {
    0,    0,    0,    0,    0,    0,    0,    0
,   0,    0,    0,    0,    0,    0,    0,    0
,0xf3,0xff,0xc7,0x70,0x3f,0x94,0x00,0xe5
,0x2a,0x7d,0xfb,0x4b,0x3d,0x33,0x05,0xd9
,0x8e,0x99,0x3b,0x9f,0x48,0x68,0x12,0x73
,0xc2,0x96,0x50,0xba,0x32,0xfc,0x76,0xce
,0x48,0x33,0x2e,0xa7,0x16,0x4d,0x96,0xa4
,0x47,0x6f,0xb8,0xc5,0x31,0xa1,0x18,0x6a
,0xc0,0xdf,0xc1,0x7c,0x98,0xdc,0xe8,0x7b
,0x4d,0xa7,0xf0,0x11,0xec,0x48,0xc9,0x72
,0x71,0xd2,0xc2,0x0f,0x9b,0x92,0x8f,0xe2
,0x27,0x0d,0x6f,0xb8,0x63,0xd5,0x17,0x38
,0xb4,0x8e,0xee,0xe3,0x14,0xa7,0xcc,0x8a
,0xb9,0x32,0x16,0x45,0x48,0xe5,0x26,0xae
,0x90,0x22,0x43,0x68,0x51,0x7a,0xcf,0xea
,0xbd,0x6b,0xb3,0x73,0x2b,0xc0,0xe9,0xda
,0x99,0x83,0x2b,0x61,0xca,0x01,0xb6,0xde
,0x56,0x24,0x4a,0x9e,0x88,0xd5,0xf9,0xb3
,0x79,0x73,0xf6,0x22,0xa4,0x3d,0x14,0xa6
,0x59,0x9b,0x1f,0x65,0x4c,0xb4,0x5a,0x74
,0xe3,0x55,0xa5
} ;

unsigned char m[163];

main()
{
  int i;
  if (crypto_box_curve25519xsalsa20poly1305_open(
        m,c,163,nonce,alicepk,bobsk
      ) == 0) {
    for (i = 32;i < 163;++i) {
      printf(",0x%02x",(unsigned int) m[i]);
      if (i % 8 == 7) printf("\n");
    }
    printf("\n");
  }
  return 0;
}
```

The program prints the original 131-byte packet:

```
,0xbe,0x07,0x5f,0xc5,0x3c,0x81,0xf2,0xd5
,0xcf,0x14,0x13,0x16,0xeb,0xeb,0x0c,0x7b
,0x52,0x28,0xc5,0x2a,0x4c,0x62,0xcb,0xd4
,0x4b,0x66,0x84,0x9b,0x64,0x24,0x4f,0xfc
,0xe5,0xec,0xba,0xaf,0x33,0xbd,0x75,0x1a
,0x1a,0xc7,0x28,0xd4,0x5e,0x6c,0x61,0x29
,0x6c,0xdc,0x3c,0x01,0x23,0x35,0x61,0xf4
,0x1d,0xb6,0x6c,0xce,0x31,0x4a,0xdb,0x31
,0x0e,0x3b,0xe8,0x25,0x0c,0x46,0xf0,0x6d
,0xce,0xea,0x3a,0x7f,0xa1,0x34,0x80,0x57
,0xe2,0xf6,0x55,0x6a,0xd6,0xb1,0x31,0x8a
,0x02,0x4a,0x83,0x8f,0x21,0xaf,0x1f,0xde
,0x04,0x89,0x77,0xeb,0x48,0xf5,0x9f,0xfd
,0x49,0x24,0xca,0x1c,0x60,0x90,0x2e,0x52
,0xf0,0xa0,0x89,0xbc,0x76,0x89,0x70,0x40
```

```
,0xe0,0x82,0xf9,0x37,0x76,0x38,0x48,0x64
,0x5e,0x07,0x05
```

Testing: `box` **vs.** `secretbox`**.** The following program computes the same 147-byte boxed packet, but starts from the first-level key k_1 computed in Chapter 22:

```
#include <stdio.h>
#include "crypto_secretbox_xsalsa20poly1305.h"

unsigned char firstkey[32] = {
 0x1b,0x27,0x55,0x64,0x73,0xe9,0x85,0xd4
,0x62,0xcd,0x51,0x19,0x7a,0x9a,0x46,0xc7
,0x60,0x09,0x54,0x9e,0xac,0x64,0x74,0xf2
,0x06,0xc4,0xee,0x08,0x44,0xf6,0x83,0x89
} ;

unsigned char nonce[24] = {
 0x69,0x69,0x6e,0xe9,0x55,0xb6,0x2b,0x73
,0xcd,0x62,0xbd,0xa8,0x75,0xfc,0x73,0xd6
,0x82,0x19,0xe0,0x03,0x6b,0x7a,0x0b,0x37
} ;

// API requires first 32 bytes to be 0
unsigned char m[163] = {
    0,   0,   0,   0,   0,   0,   0,   0
,   0,   0,   0,   0,   0,   0,   0,   0
,   0,   0,   0,   0,   0,   0,   0,   0
,   0,   0,   0,   0,   0,   0,   0,   0
,0xbe,0x07,0x5f,0xc5,0x3c,0x81,0xf2,0xd5
,0xcf,0x14,0x13,0x16,0xeb,0xeb,0x0c,0x7b
,0x52,0x28,0xc5,0x2a,0x4c,0x62,0xcb,0xd4
,0x4b,0x66,0x84,0x9b,0x64,0x24,0x4f,0xfc
,0xe5,0xec,0xba,0xaf,0x33,0xbd,0x75,0x1a
,0x1a,0xc7,0x28,0xd4,0x5e,0x6c,0x61,0x29
,0x6c,0xdc,0x3c,0x01,0x23,0x35,0x61,0xf4
,0x1d,0xb6,0x6c,0xce,0x31,0x4a,0xdb,0x31
,0x0e,0x3b,0xe8,0x25,0x0c,0x46,0xf0,0x6d
,0xce,0xea,0x3a,0x7f,0xa1,0x34,0x80,0x57
,0xe2,0xf6,0x55,0x6a,0xd6,0xb1,0x31,0x8a
,0x02,0x4a,0x83,0x8f,0x21,0xaf,0x1f,0xde
,0x04,0x89,0x77,0xeb,0x48,0xf5,0x9f,0xfd
,0x49,0x24,0xca,0x1c,0x60,0x90,0x2e,0x52
,0xf0,0xa0,0x89,0xbc,0x76,0x89,0x70,0x40
,0xe0,0x82,0xf9,0x37,0x76,0x38,0x48,0x64
,0x5e,0x07,0x05
} ;
```

```
unsigned char c[163];

main()
{
  int i;
  crypto_secretbox_xsalsa20poly1305(
    c,m,163,nonce,firstkey
  );
  for (i = 16;i < 163;++i) {
    printf(",0x%02x",(unsigned int) c[i]);
    if (i % 8 == 7) printf("\n");
  }
  printf("\n");
  return 0;
}
```

The following program opens the same box, again starting from the first-level key k_1:

```
#include <stdio.h>
#include "crypto_secretbox_xsalsa20poly1305.h"

unsigned char firstkey[32] = {
 0x1b,0x27,0x55,0x64,0x73,0xe9,0x85,0xd4
,0x62,0xcd,0x51,0x19,0x7a,0x9a,0x46,0xc7
,0x60,0x09,0x54,0x9e,0xac,0x64,0x74,0xf2
,0x06,0xc4,0xee,0x08,0x44,0xf6,0x83,0x89
} ;

unsigned char nonce[24] = {
 0x69,0x69,0x6e,0xe9,0x55,0xb6,0x2b,0x73
,0xcd,0x62,0xbd,0xa8,0x75,0xfc,0x73,0xd6
,0x82,0x19,0xe0,0x03,0x6b,0x7a,0x0b,0x37
} ;

// API requires first 16 bytes to be 0
unsigned char c[163] = {
    0,   0,   0,   0,   0,   0,   0,   0
,   0,   0,   0,   0,   0,   0,   0,   0
,0xf3,0xff,0xc7,0x70,0x3f,0x94,0x00,0xe5
,0x2a,0x7d,0xfb,0x4b,0x3d,0x33,0x05,0xd9
,0x8e,0x99,0x3b,0x9f,0x48,0x68,0x12,0x73
,0xc2,0x96,0x50,0xba,0x32,0xfc,0x76,0xce
,0x48,0x33,0x2e,0xa7,0x16,0x4d,0x96,0xa4
,0x47,0x6f,0xb8,0xc5,0x31,0xa1,0x18,0x6a
,0xc0,0xdf,0xc1,0x7c,0x98,0xdc,0xe8,0x7b
,0x4d,0xa7,0xf0,0x11,0xec,0x48,0xc9,0x72
,0x71,0xd2,0xc2,0x0f,0x9b,0x92,0x8f,0xe2
,0x27,0x0d,0x6f,0xb8,0x63,0xd5,0x17,0x38
,0xb4,0x8e,0xee,0xe3,0x14,0xa7,0xcc,0x8a
,0xb9,0x32,0x16,0x45,0x48,0xe5,0x26,0xae
,0x90,0x22,0x43,0x68,0x51,0x7a,0xcf,0xea
,0xbd,0x6b,0xb3,0x73,0x2b,0xc0,0xe9,0xda
,0x99,0x83,0x2b,0x61,0xca,0x01,0xb6,0xde
,0x56,0x24,0x4a,0x9e,0x88,0xd5,0xf9,0xb3
,0x79,0x73,0xf6,0x22,0xa4,0x3d,0x14,0xa6
,0x59,0x9b,0x1f,0x65,0x4c,0xb4,0x5a,0x74
,0xe3,0x55,0xa5
} ;

unsigned char m[163];

main()
{
  int i;
  if (crypto_secretbox_xsalsa20poly1305_open(
        m,c,163,nonce,firstkey
      ) == 0) {
    for (i = 32;i < 163;++i) {
      printf(",0x%02x",(unsigned int) m[i]);
      if (i % 8 == 7) printf("\n");
    }
    printf("\n");
  }
  return 0;
}
```

Testing: `secretbox` **vs.** `stream`**.** The following program starts from the first-level key k_1 shown above, computes the first 163 bytes of the corresponding stream as in Chapter 22, skips the first 32 bytes, and xors the remaining bytes with the 131-byte packet shown above:

```
#include <stdio.h>
#include "crypto_stream_xsalsa20.h"

unsigned char firstkey[32] = {
 0x1b,0x27,0x55,0x64,0x73,0xe9,0x85,0xd4
,0x62,0xcd,0x51,0x19,0x7a,0x9a,0x46,0xc7
,0x60,0x09,0x54,0x9e,0xac,0x64,0x74,0xf2
,0x06,0xc4,0xee,0x08,0x44,0xf6,0x83,0x89
} ;

unsigned char nonce[24] = {
 0x69,0x69,0x6e,0xe9,0x55,0xb6,0x2b,0x73
,0xcd,0x62,0xbd,0xa8,0x75,0xfc,0x73,0xd6
,0x82,0x19,0xe0,0x03,0x6b,0x7a,0x0b,0x37
} ;
```

```
unsigned char m[163] = {
    0,   0,   0,   0,   0,   0,   0,   0
,   0,   0,   0,   0,   0,   0,   0,   0
,   0,   0,   0,   0,   0,   0,   0,   0
,   0,   0,   0,   0,   0,   0,   0,   0
,0xbe,0x07,0x5f,0xc5,0x3c,0x81,0xf2,0xd5
,0xcf,0x14,0x13,0x16,0xeb,0xeb,0x0c,0x7b
,0x52,0x28,0xc5,0x2a,0x4c,0x62,0xcb,0xd4
,0x4b,0x66,0x84,0x9b,0x64,0x24,0x4f,0xfc
,0xe5,0xec,0xba,0xaf,0x33,0xbd,0x75,0x1a
,0x1a,0xc7,0x28,0xd4,0x5e,0x6c,0x61,0x29
,0x6c,0xdc,0x3c,0x01,0x23,0x35,0x61,0xf4
,0x1d,0xb6,0x6c,0xce,0x31,0x4a,0xdb,0x31
,0x0e,0x3b,0xe8,0x25,0x0c,0x46,0xf0,0x6d
,0xce,0xea,0x3a,0x7f,0xa1,0x34,0x80,0x57
,0xe2,0xf6,0x55,0x6a,0xd6,0xb1,0x31,0x8a
,0x02,0x4a,0x83,0x8f,0x21,0xaf,0x1f,0xde
,0x04,0x89,0x77,0xeb,0x48,0xf5,0x9f,0xfd
,0x49,0x24,0xca,0x1c,0x60,0x90,0x2e,0x52
,0xf0,0xa0,0x89,0xbc,0x76,0x89,0x70,0x40
,0xe0,0x82,0xf9,0x37,0x76,0x38,0x48,0x64
,0x5e,0x07,0x05
} ;

unsigned char c[163];

main()
{
  int i;
  crypto_stream_xsalsa20_xor(c,m,163,nonce,firstkey);
  for (i = 32;i < 163;++i) {
    printf(",0x%02x",(unsigned int) c[i]);
    if (i % 8 == 7) printf("\n");
  }
  printf("\n");
  return 0;
}
```

This program prints

```
,0x8e,0x99,0x3b,0x9f,0x48,0x68,0x12,0x73
,0xc2,0x96,0x50,0xba,0x32,0xfc,0x76,0xce
,0x48,0x33,0x2e,0xa7,0x16,0x4d,0x96,0xa4
,0x47,0x6f,0xb8,0xc5,0x31,0xa1,0x18,0x6a
,0xc0,0xdf,0xc1,0x7c,0x98,0xdc,0xe8,0x7b
,0x4d,0xa7,0xf0,0x11,0xec,0x48,0xc9,0x72
,0x71,0xd2,0xc2,0x0f,0x9b,0x92,0x8f,0xe2
```

```
,0x27,0x0d,0x6f,0xb8,0x63,0xd5,0x17,0x38
,0xb4,0x8e,0xee,0xe3,0x14,0xa7,0xcc,0x8a
,0xb9,0x32,0x16,0x45,0x48,0xe5,0x26,0xae
,0x90,0x22,0x43,0x68,0x51,0x7a,0xcf,0xea
,0xbd,0x6b,0xb3,0x73,0x2b,0xc0,0xe9,0xda
,0x99,0x83,0x2b,0x61,0xca,0x01,0xb6,0xde
,0x56,0x24,0x4a,0x9e,0x88,0xd5,0xf9,0xb3
,0x79,0x73,0xf6,0x22,0xa4,0x3d,0x14,0xa6
,0x59,0x9b,0x1f,0x65,0x4c,0xb4,0x5a,0x74
,0xe3,0x55,0xa5
```

matching the final 131 bytes of the 147-byte boxed packet shown above.

Testing: `secretbox` vs. `onetimeauth`. The following program starts from the first-level key k_1 shown above and prints the first 32 bytes of the corresponding stream:

```
#include <stdio.h>
#include "crypto_stream_xsalsa20.h"

unsigned char firstkey[32] = {
 0x1b,0x27,0x55,0x64,0x73,0xe9,0x85,0xd4
,0x62,0xcd,0x51,0x19,0x7a,0x9a,0x46,0xc7
,0x60,0x09,0x54,0x9e,0xac,0x64,0x74,0xf2
,0x06,0xc4,0xee,0x08,0x44,0xf6,0x83,0x89
} ;

unsigned char nonce[24] = {
 0x69,0x69,0x6e,0xe9,0x55,0xb6,0x2b,0x73
,0xcd,0x62,0xbd,0xa8,0x75,0xfc,0x73,0xd6
,0x82,0x19,0xe0,0x03,0x6b,0x7a,0x0b,0x37
} ;

unsigned char rs[32];

main()
{
  int i;
  crypto_stream_xsalsa20(rs,32,nonce,firstkey);
  for (i = 0;i < 32;++i) {
    printf(",0x%02x",(unsigned int) rs[i]);
    if (i % 8 == 7) printf("\n");
  }
  return 0;
}
```

The output of the program is a Poly1305 key (r, s):

```
,0xee,0xa6,0xa7,0x25,0x1c,0x1e,0x72,0x91
,0x6d,0x11,0xc2,0xcb,0x21,0x4d,0x3c,0x25
```

```
,0x25,0x39,0x12,0x1d,0x8e,0x23,0x4e,0x65
,0x2d,0x65,0x1f,0xa4,0xc8,0xcf,0xf8,0x80
```

The following program starts from this Poly1305 key (r, s) and the 131-byte suffix c of the boxed packet shown above, and uses C NaCl to compute Poly1305(ClampP$(r), c, s)$:

```
#include <stdio.h>
#include "crypto_onetimeauth_poly1305.h"

unsigned char rs[32] = {
 0xee,0xa6,0xa7,0x25,0x1c,0x1e,0x72,0x91
,0x6d,0x11,0xc2,0xcb,0x21,0x4d,0x3c,0x25
,0x25,0x39,0x12,0x1d,0x8e,0x23,0x4e,0x65
,0x2d,0x65,0x1f,0xa4,0xc8,0xcf,0xf8,0x80
} ;

unsigned char c[131] = {
 0x8e,0x99,0x3b,0x9f,0x48,0x68,0x12,0x73
,0xc2,0x96,0x50,0xba,0x32,0xfc,0x76,0xce
,0x48,0x33,0x2e,0xa7,0x16,0x4d,0x96,0xa4
,0x47,0x6f,0xb8,0xc5,0x31,0xa1,0x18,0x6a
,0xc0,0xdf,0xc1,0x7c,0x98,0xdc,0xe8,0x7b
,0x4d,0xa7,0xf0,0x11,0xec,0x48,0xc9,0x72
,0x71,0xd2,0xc2,0x0f,0x9b,0x92,0x8f,0xe2
,0x27,0x0d,0x6f,0xb8,0x63,0xd5,0x17,0x38
,0xb4,0x8e,0xee,0xe3,0x14,0xa7,0xcc,0x8a
,0xb9,0x32,0x16,0x45,0x48,0xe5,0x26,0xae
,0x90,0x22,0x43,0x68,0x51,0x7a,0xcf,0xea
,0xbd,0x6b,0xb3,0x73,0x2b,0xc0,0xe9,0xda
,0x99,0x83,0x2b,0x61,0xca,0x01,0xb6,0xde
,0x56,0x24,0x4a,0x9e,0x88,0xd5,0xf9,0xb3
,0x79,0x73,0xf6,0x22,0xa4,0x3d,0x14,0xa6
,0x59,0x9b,0x1f,0x65,0x4c,0xb4,0x5a,0x74
,0xe3,0x55,0xa5
} ;

unsigned char a[16];

main()
{
  int i;
  crypto_onetimeauth_poly1305(a,c,131,rs);
  for (i = 0;i < 16;++i) {
    printf(",0x%02x",(unsigned int) a[i]);
    if (i % 8 == 7) printf("\n");
  }
  return 0;
}
```

The program prints

```
,0xf3,0xff,0xc7,0x70,0x3f,0x94,0x00,0xe5
,0x2a,0x7d,0xfb,0x4b,0x3d,0x33,0x05,0xd9
```

matching the first 16 bytes of the boxed packet shown above.

Testing: C++ vs. `onetimeauth`. The following C++ program starts from the same Poly1305 key (r, s) and the same c as above, and uses GMP (through GMP's C++ interface) to compute Poly1305(ClampP$(r), c, s$):

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

void poly1305_gmpxx(unsigned char *out,
  const unsigned char *r,
  const unsigned char *s,
  const unsigned char *m,unsigned int l)
{
  unsigned int j;
  mpz_class rbar = 0;
  for (j = 0;j < 16;++j) {
    mpz_class rj = r[j];
    if (j % 4 == 3) rj = r[j] % 16;
    if (j == 4) rj = r[j] & 252;
    if (j == 8) rj = r[j] & 252;
    if (j == 12) rj = r[j] & 252;
    rbar += rj << (8 * j);
  }
  mpz_class h = 0;
  mpz_class p = (((mpz_class) 1) << 130) - 5;
  while (l > 0) {
    mpz_class c = 0;
    for (j = 0;(j < 16) && (j < l);++j)
      c += ((mpz_class) m[j]) << (8 * j);
    c += ((mpz_class) 1) << (8 * j);
    m += j; l -= j;
    h = ((h + c) * rbar) % p;
  }
  for (j = 0;j < 16;++j)
    h += ((mpz_class) s[j]) << (8 * j);
  for (j = 0;j < 16;++j) {
    mpz_class c = h % 256;
    h >>= 8;
    out[j] = c.get_ui();
  }
}

unsigned char rs[32] = {
 0xee,0xa6,0xa7,0x25,0x1c,0x1e,0x72,0x91
,0x6d,0x11,0xc2,0xcb,0x21,0x4d,0x3c,0x25
,0x25,0x39,0x12,0x1d,0x8e,0x23,0x4e,0x65
,0x2d,0x65,0x1f,0xa4,0xc8,0xcf,0xf8,0x80
} ;

unsigned char c[131] = {
 0x8e,0x99,0x3b,0x9f,0x48,0x68,0x12,0x73
,0xc2,0x96,0x50,0xba,0x32,0xfc,0x76,0xce
,0x48,0x33,0x2e,0xa7,0x16,0x4d,0x96,0xa4
,0x47,0x6f,0xb8,0xc5,0x31,0xa1,0x18,0x6a
,0xc0,0xdf,0xc1,0x7c,0x98,0xdc,0xe8,0x7b
,0x4d,0xa7,0xf0,0x11,0xec,0x48,0xc9,0x72
,0x71,0xd2,0xc2,0x0f,0x9b,0x92,0x8f,0xe2
,0x27,0x0d,0x6f,0xb8,0x63,0xd5,0x17,0x38
,0xb4,0x8e,0xee,0xe3,0x14,0xa7,0xcc,0x8a
,0xb9,0x32,0x16,0x45,0x48,0xe5,0x26,0xae
,0x90,0x22,0x43,0x68,0x51,0x7a,0xcf,0xea
,0xbd,0x6b,0xb3,0x73,0x2b,0xc0,0xe9,0xda
,0x99,0x83,0x2b,0x61,0xca,0x01,0xb6,0xde
,0x56,0x24,0x4a,0x9e,0x88,0xd5,0xf9,0xb3
,0x79,0x73,0xf6,0x22,0xa4,0x3d,0x14,0xa6
,0x59,0x9b,0x1f,0x65,0x4c,0xb4,0x5a,0x74
,0xe3,0x55,0xa5
} ;

unsigned char a[16];

main()
{
  int i;
  poly1305_gmpxx(a,rs,rs + 16,c,131);
  for (i = 0;i < 16;++i) {
    printf(",0x%02x",(unsigned int) a[i]);
    if (i % 8 == 7) printf("\n");
  }
  return 0;
}
```

The program prints

```
,0xf3,0xff,0xc7,0x70,0x3f,0x94,0x00,0xe5
,0x2a,0x7d,0xfb,0x4b,0x3d,0x33,0x05,0xd9
```

matching the output of the previous program.

Chapter 25

Signatures

This chapter briefly describes the `crypto_sign_edwards25519sha512batch` signature system supported by NaCl.

The system parameters are an elliptic curve in twisted Edwards form $-x^2+y^2 = 1+dx^2y^2$ over the finite field $\mathbf{F}_p$, where $p = 2^{255} - 19$ and d is chosen such that the curve is birationally equivalent to Curve25519; a point $P \in E(\mathbf{F}_p)$ of prime order p_1; and a cryptographic hash function H, which is chosen to be SHA-512.

The long term secret key of user A is a tuple of integers (a_1, a_2); the corresponding public key is $P_A = a_1P$. The second part of the secret key is used in nonce generation.

To sign message m, user A first computes a message-dependent nonce $k = H(a_2||m)$ and then computes $R = kP$. The compressed representation r of R consists of the y-coordinate of R and one bit of the x coordinate to distinguish R from $-R$. The signature consists of $(r, s) = (r, H(r||m)k + a_1)$, where the second part is computed modulo p_1.

To verify that a signature (r, s) on a message m belongs to the user with public key P_A the recipient decompresses r into a point R on the Edwards curve, computes $h = H(r||m)$ and checks whether $hR + P_A$ equals sP. If so, the signature is accepted as valid; otherwise it is rejected as invalid. Signatures from A are accepted as valid since $sP = (H(r||m)k+a_1)P = H(r||m)(kP) + a_1P = hR + P_A$.

Signature verification is computed more efficiently as a double scalar multiplication, checking whether $sP - hR$ equals P_A. This equation lends itself to batch verification, for example saving about a factor of 4 in verifying 100 signatures simultaneously; see [34] and [14].

This signature scheme has several efficiency and security advantages over the standard elliptic-curve signature scheme ECDSA [26]:

- Signature generation and verification do not use inversion modulo the group order p_1. This speeds up the computation and simplifies implementations.
- Signatures can be verified in batches for higher efficiency.
- Different messages are guaranteed to receive different, and unpredictable, nonces.
- All uses of H are randomized, so collisions in H have no obvious impact on security.
- There is no obvious malleability of signatures. NaCl does not promise strong unforgeability (non-malleability) for signatures in general, but this feature might be useful if there are any applications of strong unforgeability.

Part III:

Networking in NaCl: the CurveCP protocol

Chapter 26

Introduction to networking

Sending data through the Internet is like sending it through the radio (and often *is* actually sending it through the radio). An attacker who sets up a radio nearby can spy on everything that you're sending, and on everything that you're receiving. Some, perhaps most, of the data you send and receive is public, but the attacker can also see all the private information.

The attacker doesn't have to be physically close to you. He could be halfway around the world. He simply has to have control over *one* computer on the same network as yours, or on the same network as the computer you're talking to, or on any network in between. Do you really think that these hundreds of computers are all trustworthy and secure and not spying on your communication?

Furthermore, the attacker can forge packets of data that look like they come from you, or from whichever site you're talking to. With slightly more effort the attacker can spy on each packet as you're sending it, quickly substitute a modified version of the packet, and prevent the receiver from seeing the original packet.

26.1 NaCl and CurveCP

CurveCP is a new networking protocol described in the following chapters. CurveCP is similar to TCP but uses high-speed high-security elliptic-curve cryptography to protect every packet against espionage, corruption, and sabotage. NaCl's networking component is a complete implementation of CurveCP.

Chapter 27

Confidentiality and integrity

The CurveCP client and CurveCP server encrypt and authenticate each packet, scrambling the packet in a way that only they can create and only they can understand. This cryptographic protection provides **confidentiality** against espionage and **integrity** against corruption and forgery. CurveCP also provides some **availability** against sabotage; this is discussed in the next chapter.

The client and server administrators don't meet secretly to share an encryption/authentication key. The server has a long-term public key S visible to everyone; the client uses this key S to encrypt data for the server and to verify data from the server. The client also has a long-term public key C visible to the server; the server uses that key C to encrypt data for the client and to verify data from the client.

27.1 Does CurveCP provide server authentication?

Yes. The client verifies every packet from the server using the server's long-term public key, or using a short-term public key that the long-term key has securely vouched for. Attackers cannot pretend to be the server; forged packets labelled as being from the server are immediately recognized and discarded.

27.2 Does CurveCP provide client authentication?

Yes. The client's long-term public key securely vouches for a short-term public key, and the server verifies every packet from the client using that short-term public key. Attackers cannot pretend to be the client; forged packets labelled as being from the client are immediately recognized and discarded.

27.3 Does CurveCP stop replay attacks?

Yes. If the attacker makes copies of a legitimate client's Hello packets then the attacker will receive server Cookie packets without affecting the server state; these Cookie packets do not leak information and will be rejected by the legitimate client. If the attacker makes copies of other client packets then the copies will be rejected by this server and by other servers. If the attacker makes copies of server packets then the copies will be rejected by this client and by other clients.

27.4 Does CurveCP stop man-in-the-middle attacks?

Yes. The client knows the server's long-term public key in advance, before making a CurveCP connection. Servers that differentiate between clients know the clients' long-term public keys in advance. An attacker is then unable to

- fool the client into encrypting data to the attacker rather than the server;
- fool the server into accepting data from the attacker and thinking it comes from the client;
- fool the server into sending responses to the attacker rather than the client; or
- fool the client into accepting data from the attacker and thinking it comes from the server.

CurveCP's encryption mechanism is tied tightly and securely to its client/server identification and authentication mechanisms. An attacker with full control over every packet between the client and server can compromise availability but cannot compromise confidentiality or integrity.

27.5 Does CurveCP provide forward secrecy?

Yes. Two minutes after a connection is closed, both the client and the server are unable to understand (or verify) what was sent through the network. Of course, the client and server might have stored copies of the plaintext on disk, but that isn't CurveCP's fault.

CurveCP also provides forward secrecy for the client's long-term public key. Two minutes after a connection is closed, the server is unable to extract the client's long-term public key from the network packets that were sent to that server, and is unable to verify the client's long-term public key from the network packets.

Here's how the forward secrecy works. At the beginning of a connection, the CurveCP server generates a short-term public key S' and short-term secret key s', supplementing its long-term public key S and long-term secret key s. Similarly, the CurveCP client generates its own short-term public key C' and short-term secret key c', supplementing its long-term public key C and long-term secret key c. Almost all components of CurveCP packets are in cryptographic boxes that can be opened only by the short-term secret keys s' and c'. The only exceptions are as follows:

- Packets from the client contain, unencrypted, the short-term public key C'. This public key is generated randomly for this CurveCP connection; it is tied to the connection but does not leak any other information.
- The first packet from the client contains a cryptographic box that can be opened by c' and by s (not s'; the client does not know S' at this point). However, this box contains nothing other than constant padding.
- The first packet from the server contains a cryptographic box that can be opened by c' and by s. However, this box contains nothing other than the server's short-term public key S', which is generated randomly for this CurveCP connection, and a cookie, discussed below.

- The second packet from the client contains a cookie from the server. This cookie is actually a cryptographic box that can be understood only by a "minute key" in the server. Two minutes later the server has discarded this key and is unable to extract any information from the cookie.

At the end of the connection, both sides throw away the short-term secret keys s' and c'.

27.6 Does CurveCP provide options to disable encryption and to disable authentication?

No. CurveCP's server authentication is always active and cannot be disabled. CurveCP's client authentication is always active and cannot be disabled. CurveCP's encryption is always active and cannot be disabled. CurveCP's forward secrecy is always active and cannot be disabled. CurveCP has nothing analogous to IPsec's separation between AH and ESP, and nothing analogous to HTTPS renegotiation.

The lack of options in CurveCP simplifies the protocol and prevents a wide range of design and implementation mistakes. Compare the following quotes:

> Ferguson and Schneier, 2003: Our main criticism of IPsec is its complexity. IPsec contains too many options and too much flexibility; there are often several ways of doing the same or similar things. This is a typical committee effect. Committees are notorious for adding features, options, and additional flexibility to satisfy various factions within the committee. As we all know, this additional complexity and bloat is seriously detrimental to a normal (functional) standard. However, it has a devastating effect on a security standard. ... When both encryption and authentication are provided, IPsec performs the encryption first, and authenticates the ciphertext. In our opinion, this is the wrong order. ... Authentication should thus be applied to the plaintext (as it is in SSL [FKK96]), and not to the ciphertext. ... An Attack on IPsec ... Clearly, the authentication property has been violated.

> Degabriele and Paterson, 2010: IPsec allows a huge amount of flexibility in the ways in which its component cryptographic mechanisms can be combined to build a secure communications service. This may be good for supporting different security requirements but is potentially bad for security. We demonstrate the reality of this by describing efficient, plaintext-recovering attacks against all configurations of IPsec in which integrity protection is applied prior to encryption – so-called MAC-then-encrypt configurations. ... For concreteness, we study the common use case of using IPsec to build a simple site-to-site VPN. We describe practical attacks against all MAC-then-encrypt configurations of IPsec for this common application, including the most natural configurations as well as more "exotic" ones.

Ferguson and Schneier break an encrypt-then-MAC configuration for IPsec and recommend MAC-then-encrypt. Degabriele and Paterson show that IPsec's MAC-then-encrypt options are completely insecure. The only point of agreement between the authors is that committee-induced flexibility is a disaster for security.

27.7 Does CurveCP provide client address authentication?

No. IP addresses are not secure in any meaningful sense, and CurveCP does not attempt to make them secure. Servers that distinguish between clients must do so on the basis of long-term client public keys, not IP addresses.

27.8 Does CurveCP protect against traffic analysis?

No. An attacker can see the Internet destination, exact timing, and approximate length of each packet that you send. CurveCP is not a substitute for an anonymizing network such as Tor, although of course it can be used to protect links inside an anonymizing network.

Don't underestimate the amount of interesting information that the attacker can deduce from traffic analysis! For example, Song, Wagner, and Tian showed in 2001 that the timing of password packets in ssh leaks several bits of information about a typical password.

Chapter 28

Availability

CurveCP takes several steps to improve Internet availability, i.e., to make denial-of-service attacks more difficult.

28.1 The problem, part 1: availability is more than integrity

Alice sends data to Bob: web pages, email messages, etc. What data does Bob think Alice has sent?

Integrity ("the truth and nothing but the truth") means that all of the data that Bob thinks Alice sent was, in fact, sent to Bob by Alice, not by an attacker.

Availability ("the truth, *the whole truth*, and nothing but the truth") means that the data that Bob thinks Alice sent is exactly the data that Alice actually sent to Bob: nothing more *and nothing less.*

For example, an attacker can abort an SSH connection or an HTTPS connection by forging a single TCP Reset packet. The user sees an error message such as "Connection closed by remote host" or "Connection reset by peer." The user isn't receiving bad data from the attacker, but also isn't receiving good data from the legitimate server. This is not a violation of integrity but it is a violation of availability.

As another example, PGP-encrypting and PGP-signing an email message protects confidentiality and integrity: an attacker who steals the email message won't understand it and can't replace it with a different message. But PGP does nothing to protect availability: the email has silently disappeared! Retroactively checking integrity can't restore availability.

28.2 The problem, part 2: availability is quantitative

Alice and Bob can communicate through a completely hostile network, filled with billions of attack computers, and still be assured of the confidentiality and integrity of their data. Modern cryptographic techniques reach levels of confidentiality and integrity that are infeasible to break.

The same is not true for availability. For example, a large botnet sending a flood of packets will overwhelm the network between Alice and Bob, preventing communication between Alice and Bob; or a malicious ISP can simply refuse to deliver packets from Alice to Bob; or an attacker with physical access to the network can cut a cable. These attacks do not violate confidentiality or integrity, but they do violate availability.

Internet designers can nevertheless strive to *improve* availability by *increasing* the cost of denying service and *decreasing* the amount of communication affected by denial of service. CurveCP takes several steps in this direction.

Beware that there are also serious denial-of-service problems in many other layers of Internet communication, including the physical layer (especially 802.11) and the routing layer (BGP). CurveCP eliminates several denial-of-service problems in the transport layer (TCP), but much more work has to be done.

28.3 Unauthenticated packets

Let's look at Bob copying a tiny file through SSH. Measurements show that SSH uses a total of 40 packets back and forth to set up the connection and copy the file.

An attacker eavesdropping on the network watches Bob set up the connection. After (say) 20 packets the attacker forges a TCP Reset packet, or a regular TCP packet containing bogus data. Bob's connection dies. Bob has not yet received any of the legitimate data. If Bob tries again, the attacker breaks that connection too. The attacker is sending only a tiny fraction as much data as Bob is sending, and is still preventing Bob from communicating.

CurveCP is different. CurveCP authenticates and verifies *each* packet. If a bogus packet arrives, CurveCP simply discards the packet and waits for the legitimate packet.

An attacker can, at somewhat higher cost, prevent the legitimate packet from being delivered. But CurveCP handles this in the same way as other network failures: it recognizes that data is missing and retransmits the data. To stop communication the attacker has to stop essentially *all* of the legitimate packets; this is considerably more expensive for the attacker than forging an occasional packet, so the number of victims is reduced.

28.4 Predictable retransmission times

An attacker who watches the client send a few packets in a typical TCP connection can easily predict when the client will send its next packet. The attacker can stop that packet by briefly flooding the network *at that moment.* The attacker can also predict when TCP will try retransmitting the packet, and can stop that retransmission with another carefully timed flood. This is less expensive than flooding the network continuously.

CurveCP adds some randomness to its packet-transmission schedule. This randomness gives each packet a chance of fitting through the holes in any part-time flood pattern chosen by the attacker.

28.5 Blind amplification

Some protocols allow attackers anywhere on the Internet to generate a packet that will trigger a much larger outgoing packet from the server to a victim address selected by the attacker. The issue here isn't the availability of that protocol; the issue is that the protocol is amplifying the attacker's resources, damaging availability for the rest of the Internet. The worst offender at the moment is DNSSEC, which has set up a remote-controlled machine-gun pool containing more than 2000 servers with amplification factors between 30 and 95 and with an overall outgoing attack capacity estimated to be close to 50 gigabits per second. See [17].

With CurveCP, the first incoming packet from the client is padded so that it is as large as the outgoing packet from the server. If this padding is missing, the server won't respond. Subsequent packets from the client need to repeat server cookies and can't be generated blindly.

28.6 Non-blind amplification

Heavy amplification sometimes hurts availability even when it's limited to attackers who can see where the packets are going. An attacker who has seized control over a few computers on a network can use those computers to generate a much larger flood back to the same network.

Insisting on all client packets being as large as server packets would double the network bandwidth used by large downloads and still wouldn't eliminate amplification: a server that doesn't receive client responses will retransmit packets. CurveCP nevertheless takes several steps to *reduce* amplification:

- The server doesn't retransmit its first packet, the Cookie packet. The client is responsible for repeating its Hello packet to ask for another Cookie packet.
- The ratio between maximum and minimum packet sizes is much smaller in CurveCP than it is in other protocols. CurveCP packets are limited to 1024 bytes of data (plus some overhead). CurveCP never triggers packet fragmentation on standard networks.
- CurveCP backs off more quickly than TCP from a congested network, reducing the bandwidth used by retransmissions.

28.7 Blind memory consumption

Once upon a time, TCP servers would allocate memory (inside a "TCP control-block table") for each incoming SYN packet. This allowed attackers to trivially fill up memory with blind "SYN flooding": sending a stream of SYN packets.

Phil Karn introduced a cookie exchange as the first step of the Photuris protocol (1994) to prevent the attacker from *blindly* allocating memory on the server. It was later observed that TCP servers could unilaterally apply the same technique inside TCP, with no changes to TCP clients; these cookies are called "SYN cookies" (1996). The more general technique of "remote storage" eliminates storage on a server in favor of storage inside the network: the server sends data as an encrypted authenticated message to itself via the client.

A CurveCP server does not allocate memory in response to a client's first packet, the Hello packet. Instead it uses remote storage: it sends a Cookie packet containing a cookie that the client cannot understand but must repeat. This packet also contains a client-comprehensible part, the server's short-term public key S' for this connection, used to provide forward secrecy.

A CurveCP server does allocate memory in response to a client's second packet, the Initiate packet, but only if the second packet contains a valid recent cookie from the server. Blind attackers have no way to forge these cookies, and therefore no way to allocate memory on the server.

28.8 Unauthenticated memory consumption

Many SSH servers are configured as high-security “public-key-only” servers. These servers do not authorize communication from random clients; they have a list of public keys of authorized clients (in `.ssh/authorized_keys`), and are willing to talk only to clients sending data authenticated by those public keys.

An attacker can trivially fill up memory on these servers, disabling legitimate communication, by making a series of TCP connections and leaving those connections idle. SYN cookies mean that this attack cannot be carried out *blindly*, so a server can try to defend itself by imposing separate per-IP-address connection limits, but this defense is trivially subverted by attackers who control many computers or who are eavesdropping on the network.

With CurveCP, the client’s Initiate packet contains the client’s long-term public key C (encrypted to the server’s short-term public key S’), together with a message from C vouching for this connection (a Vouch subpacket). If a CurveCP server is not willing to talk to unauthorized clients, it simply checks C against the list of authorized clients, and drops the packet if C is unauthorized. Unauthorized clients therefore have no way to allocate memory on the server.

Several servers that share a long-term key still use separate minute keys, so they don’t understand each other’s cookies. An Initiate packet replayed to a different server is quickly discarded.

28.9 CPU consumption

Many cryptographic protocols allow unauthenticated attackers to easily overload server CPUs or client CPUs, preventing the CPUs from processing legitimate traffic. Of course, an attacker can achieve the same result by flooding the network; the issue here is that many protocols allow the attacker to overload the CPUs with a much smaller amount of traffic, reducing the attacker’s costs and increasing the number of victims.

There are several different mistakes that lead to this type of attack:

- Poor choice of cryptographic mechanisms. For example, TCPcrypt overloads the decryption ability of a quad-core 2.66GHz Intel Xeon X5355 client with just 400 packets per second. (TCPcrypt uses 2048-bit RSA, with security level approximately 2^{112}.) Imagine how much more trouble these computations would cause for a single-core 1GHz Apple iPad!

- Poor protocol design. For example, typical implementations of TCP SACK (selective acknowledgments) perform huge computations in response to a single forged attacker packet.

- Poor choice of data structures. For example, many servers allow hash flooding: an attacker puts a huge number of entries into a single hash-table bucket, drastically increasing the time required for the server to process those entries.

CurveCP’s worst-case server CPU load is very small, and its worst-case client CPU load is very small. CurveCP uses high-speed high-security elliptic-curve cryptography so that a typical CPU can perform public-key operations more quickly than a typical Internet connection can ask for those operations.

Chapter 29

Decongestion

The job of an Internet router is to forward packets from incoming network links (wires, radios, etc.) to outgoing network links. The router first stores each packet in an internal queue in case the outgoing link is busy. **Congestion** means that packets are continuing to arrive at the router more quickly than the outgoing link can handle them. The queue length then increases; if this persists then eventually the queue fills up and the router is forced to discard packets.

This chapter discusses the congestion-control and congestion-avoidance mechanisms in TCP and in CurveCP. These mechanisms are handled by packet schedulers that decide when to transmit packets and when to retransmit packets. There are actually several different TCP packet schedulers in common use, including CUBIC (Linux), NewReno (FreeBSD), and Compound (Microsoft). CurveCP uses a new scheduler called Chicago that decongests routers, including routers suffering from "bufferbloat"; Chicago efficiently uses the available bandwidth while minimizing packet loss and reducing latency for interactive applications.

29.1 Congestion-induced inefficiency

Congestion might at first seem to be a minor issue. Data is not permanently lost when packets are lost: clients and servers send packets again and again until the packets are acknowledged. The outgoing link will continue transmitting data at full speed—obviously the best it can do—and eventually will transmit the lost data.

The primary problem is that each lost packet wastes time on the *incoming* link. A packet sent 10 times through this link, because it was lost the first 9 times, consumes 10 times as much space as it would otherwise have consumed, effectively reducing the bandwidth of the incoming link by a factor of 10—a huge efficiency problem. Sometimes this reduction means that the *incoming* link is overloaded, congesting the *previous* router and causing even more packets to be lost.

A secondary problem is that increased queue lengths cause increased delays for packets in the queue. This is not a bandwidth problem but it is a latency problem. Users waiting for data (web pages, streaming video, etc.) frequently encounter long delays attributable directly to queue congestion, often several seconds or more.

These two problems naturally classify schedulers into three categories:

Primary evaluation (packet loss)	Secondary evaluation (latency)	Examples	Summary
Bad	Bad	TCP schedulers on the Internet until the late 1980s	Senders send packets as quickly as they can. When senders notice packet loss, they retransmit packets at high speed, causing further congestion and further packet loss.
Good	Bad	TCP schedulers on the Internet today: e.g., CUBIC (Linux), NewReno (FreeBSD), Compound (Microsoft)	Senders increase packet-sending rates until they notice packet loss as a sign of congestion. Senders then reduce packet-sending rates to keep loss rates under control. To detect increases in available network capacity (e.g., someone else's download has finished), senders continue trying higher rates, keeping queues congested and periodically causing packet loss.
Good	Good	TCP Vegas, LEDBAT (μTP in BitTorrent), Chicago (CurveCP)	Senders increase packet-sending rates until they notice packet loss *or* increased delays as a sign of congestion. Senders then adapt packet-sending rates to keep loss *and* delays under control. Queue congestion is minimized.

There are many more TCP schedulers. Most of these schedulers are in the Good+Bad line. Good+Good is obviously more desirable; the Internet's continued use of Good+Bad is discussed below.

29.2 Unfairness

Further problems appear when two or more flows (active connections) are competing for bandwidth on the same link. Users expect each flow to promptly set a **fair** rate: half of the link bandwidth when there are two flows, or 1/N of the link bandwidth when there are N flows. This requires communication between the flows.

The Internet does not, in general, provide *explicit* communication between flows. Two flows instead communicate *implicitly.* Each flow causes delays (or, more clumsily, packet losses). Presumably this signal is visible to both flows, or at least has an equal chance of being seen by each flow. Each flow separately adjusts its rate in response to this signal. If this adjustment has the effect of bringing the rates closer together, and other adjustments do not have the effect of bringing the rates farther apart, then eventually the rates will converge.

For example, many TCP schedulers use an "AIMD" adjustment mechanism that works as follows:

- Goal: The difference $R1 - R2$ will rapidly decrease towards 0. Here $R1$ and $R2$ are the two flow rates.
- "Multiplicative decrease": Each flow reduces its rate in half upon seeing a congestion signal. This chops $R1 - R2$ in half.
- "Additive increase": Each flow periodically increases its rate by a constant. This does not affect $R1 - R2$.

This is fair *if* the flows have the same idea of what "periodically" means. However, for most TCP schedulers, "periodically" is defined by the round-trip time (RTT), producing **RTT unfairness**: a flow with a faster RTT will use more of the link than a flow with a slower RTT. Some TCP schedulers schedule "periodically" on an RTT-independent scale, such as once per second, to avoid RTT unfairness; Chicago also does this.

Here is a much worse example, the **late-comer's advantage**. TCP Vegas measures the minimum RTT that it sees, and adjusts its rate so that the RTT is somewhat larger, say min+delta. Here delta is not very large (large queue delays indicate congestion), but it is also not very small (empty queues indicate an idle network). Once TCP Vegas has found a rate that keeps the RTT stably at min+delta, it does not adjust the rate further; the RTT stays at min+delta. Now suppose that one Vegas flow starts using an empty link, and then later a second Vegas flow starts using the same link. The first Vegas flow sees the RTT of the empty link as the minimum RTT, and increases its rate so that the RTT is min+delta. The second Vegas flow then arrives, sees min+delta as the minimum RTT, and quickly pushes the RTT up to min+2delta. The first Vegas flow interprets the increased RTT as a sign of competition for the network, and reduces its rate so that the RTT drops below min+2delta; the second Vegas flow then increases its rate, pushing the RTT back up to min+2delta. This continues until the rate of the first Vegas flow has converged to essentially zero. The second Vegas flow ends up monopolizing the link.

Widely deployed Good+Bad schedulers such as NewReno and CUBIC avoid this type of problem by implicitly creating a synchronized **congestion cycle**. Each cycle begins with a moment of maximum congestion (i.e., maximum queue length), decreases down to a lower level of congestion, and then increases back up to maximum congestion, ending the cycle and starting the next cycle. This congestion cycle plays a critical, and underappreciated, role as a flow-communication mechanism: each flow recognizes maximum congestion at the same moment (through packet loss) and, at that moment, decreases rate multiplicatively—enough to bring the cumulative rate below the capacity of the bottleneck link, prompting the decrease in congestion. Each flow increases rate at all other times, eventually prompting the increase in congestion.

Like NewReno and CUBIC, but unlike Vegas, Chicago uses frequent additive increases and occasional multiplicative decreases to create a synchronized congestion cycle. Unlike NewReno and CUBIC, Chicago keeps track of long-term delay statistics, and explicitly recognizes cycles that merely vary in delay, rather than requiring each cycle to end with maximum congestion and packet loss. Chicago gradually pushes the delays down, creating short cycles where the top and bottom of the cycle are at very low levels of congestion, drastically reducing latency and eliminating typical congestion-induced packet loss.

29.3 Unfriendliness

Even more problems appear when two flows *using different schedulers* are competing for bandwidth on the same link.

Here is a bad example, extreme **unfriendliness** of one TCP scheduler towards another TCP scheduler. Suppose one flow uses a widely deployed Good+Bad scheduler such as TCP CUBIC, while the other flow uses TCP Vegas. The CUBIC flow will increase its rate until it causes packet loss, filling queues and creating delays. The Vegas flow will respond to the delays by reducing its rate, while the CUBIC flow is blind to the delays. Vegas obtains bandwidth for brief moments after the CUBIC rate decreases multiplicatively, but in general uses only a very small fraction of the link. To summarize, CUBIC is extremely unfriendly to Vegas. Other Good+Bad TCP schedulers, such as NewReno, are also extremely unfriendly to Vegas. Experiments with Vegas have almost uniformly found Vegas running at extremely low speeds whenever there is even a single Good+Bad competitor. (The only exceptions are simulations of old routers using very short queues.) This is obviously dissatisfying to users who try Vegas, and appears to be a major reason for the failure of Vegas to be deployed on the Internet.

(CUBIC, NewReno, etc. are extremely unfriendly to BitTorrent's μTP LEDBAT for similar reasons. This is advertised as a *feature* of BitTorrent: if a web browser begins downloading a large web page then BitTorrent will very quickly stop using the network. However, for exactly the same reason, users who try LEDBAT for web pages will find their web browsers waiting practically forever if any other connection is using CUBIC.)

The conventional wisdom is that users will be unhappy with a new scheduler if NewReno/CUBIC/etc. are *extremely* unfriendly to that scheduler or vice versa. On the other hand, small imbalances in network usage seem much less important to users. CUBIC is *somewhat* unfriendly towards NewReno, for example, and doesn't provide RTT fairness, but these problems seem to have generated very few complaints; each connection continues to receive a tolerable share of the bandwidth, even if not a fair share.

Chicago uses **edge-triggered backoffs** so that there are only a constant number of backoffs in a typical cycle. When a Good+Bad flow runs alongside a Chicago flow, the Good+Bad flow will not notice the delays it is creating, and will not notice Chicago backing off as a result of those delays; but Chicago will then see that the cycle is continuing, and will continue increasing its rate until the actual end of the cycle. Chicago, unlike Vegas, thus receives a tolerable share of the bandwidth.

29.4 Hammering

The original TCP schedulers would begin a flow by sending every possible packet within the receiver's advertised window. This spike of traffic was often far more than a link could handle in one RTT; routers would leap from zero congestion to heavy congestion.

Modern TCP schedulers instead limit their initial transmissions (and new transmissions after some idle time) by the following algorithm. The sender transmits a single packet; then, after an RTT, two packets; then, after another RTT, four packets; then, after another RTT, eight packets; and so on. This pattern continues until a packet is lost; the scheduler then begins AIMD.

This algorithm is called **slow start** and is widely advertised as a gentle, safe way to discover the available bandwidth. However, slow start is actually quite dangerous: it can reach extremely high rates, far beyond the link capacity, placing huge spikes of traffic into router queues. If the sender continues transmitting data then the heavy congestion created by slow start will *eventually* produce packet loss, but if the sender stops before this then slow start will hammer the router queue almost as badly as the original TCP schedulers.

Chicago watches delays so that it can see when rate-doubling is beginning to create congestion.

29.5 False congestion alarms

Packets sent through wireless networks are often destroyed by radio interference. Most TCP schedulers incorrectly treat these packet losses as signs of congestion and back off multiplicatively. These schedulers make poor use of the available bandwidth: they cannot transmit more than $c/\sqrt{p}$ packets per RTT, where c is a constant depending on the scheduler and p is the packet-loss probability.

Chicago keeps track of long-term loss statistics the same way that it keeps track of long-term delay statistics. The scheduler does not confuse persistent loss with congestion-induced loss. This allows Chicago to use lossy wireless networks with reasonable efficiency.

Old TCP schedulers also misunderstood naturally occurring large delays, typically from slow modems or from delayed acknowledgments, as timeouts. Newer TCP schedulers typically use **Jacobson's algorithm** to set a timeout that takes account of variance in delays, along with either timestamps or **Karn's algorithm** to prevent miscomputation of delays for retransmitted packets. Chicago uses similar algorithms, with explicit acknowledgment of message IDs to prevent miscomputation of delays.

Chapter 30

Addressing

This chapter discusses four advantages of CurveCP over TCP:

- An ISP or site administrator can easily run a huge number of CurveCP servers on a single global IPv4 address, even if the servers are independently operated with separate long-term public keys. This feature is provided by a simple extension mechanism in CurveCP addresses.
- CurveCP servers are inherently anti-aliased, providing automatic virtual hosting and fixing some of the deficiencies in the "same-origin" policy in web browsers. This feature is provided by a simple domain-name mechanism in CurveCP addresses.
- If a site has two server addresses, and one server is down, a CurveCP client will *quickly* connect to the other address.
- A CurveCP connection remains fully functional even if the client changes IP address.

CurveCP is fully compatible with existing NAT (network address translation) mechanisms; none of the above features require clients or servers to know the global addresses of their gateways.

30.1 Server addresses

Before a client can create a CurveCP connection, it needs to know (1) the server's long-term public key and (2) the server's address. The server's address has four components:

- 4 bytes: a global IPv4 address identifying the server gateway. The client sends CurveCP packets through the Internet to this IPv4 address. Typically this address is statically configured into the gateway computer by the gateway administrator, and broadcast through DNS. (In principle, CurveCP can also be used within global address spaces other than IPv4, but this chapter focuses on IPv4 for concreteness.)
- 2 bytes: a UDP port. The client sends CurveCP packets to this UDP port. Typically this port is specified by a higher-level protocol using CurveCP, the same way that a TCP port is specified by a higher-level protocol using TCP (e.g., TCP port 25 for mail transfer through SMTP).

- 16 bytes: an extension understood by the server gateway. The semantics of this extension are decided by the server gateway administrator; the client simply copies the extension without trying to understand it.
- Variable length, at most 255 bytes: the server's domain name.

These components are typically used as follows. Packets from the client are forwarded by Internet routers until they reach the gateway computer at this IPv4 address. Normally these routers ignore the UDP port (and extension), except that some firewalls limit the clients that can reach various UDP ports. Within the gateway computer, the operating-system kernel uses the UDP port to route the packet to a CurveCP-aware program. There are two common possibilities at this point:

- Two-level gateway-server structure: The program extracts a local IPv4 or IPv6 or MAC or ... address from the CurveCP extension and forwards the packet accordingly to a local server. That local server is the CurveCP server; it has the corresponding secret key and decrypts the CurveCP packet. This last step of forwarding typically means rewriting the packet header (NAT) to contain the server's local address instead of the gateway's IPv4 address; the opposite translation happens for packets sent out to the Internet.
- Single-level structure: This program is already the CurveCP server. The server is the gateway. The server has its own IPv4 address. No extra forwarding is necessary.

The server's domain name is sent by the client (encrypted) inside the client's second packet, the Initiate packet. The server drops the Initiate packet if the domain name is wrong (i.e., does not match the server's locally configured domain name). If third parties set up unauthorized domain names that are aliased to the same server then CurveCP connections to those domain names will fail. "Virtual" servers allow several configured domain names and take different actions for different domain names.

30.2 Client addresses

The client's packets include (1) the client's short-term public key and (2) the client's address. The client's address has three components:

- 4 bytes: a global IPv4 address identifying the client gateway. The server sends CurveCP packets through the Internet to this IPv4 address. Typically this address is dynamically configured into the client, for example through a DHCP server set up by the client gateway administrator.
- 2 bytes: a UDP port. The server sends CurveCP packets to this UDP port.
- 16 bytes: an extension understood by the client gateway. The semantics of this extension are decided by the client gateway administrator; the server simply copies the extension without trying to understand it. Typically this extension is dynamically configured into the client.

Current operating-system kernels already contain facilities to automatically copy a dynamically configured IPv4 address into each outgoing UDP packet, simplifying CurveCP client software. Current kernels also assign UDP ports. Some UDP applications assign their own random UDP ports as a security feature (for kernels that don't randomize UDP ports), but this feature would be redundant for CurveCP.

Current kernels do not contain analogous facilities for extensions. Current clients read 16 bytes (binary) from the file `/etc/curvecpextension`, and re-read this file frequently to catch any updates; or use 16 all-zero bytes if that file doesn't exist.

30.3 Backup servers

An Internet server can advertise multiple IP addresses, typically the addresses of two or three servers. If one server crashes then clients can connect to another server.

Unfortunately, a typical client will start by asking TCP to connect only to the first address. If that server is unresponsive, TCP will spend a long time trying many SYN packets to that address before aborting: typically 5 packets over 180 seconds. The client will then ask TCP to connect to the second address, but the user has already given up at this point; the connection is a failure *from the user's perspective*. Typical TCP implementations don't support a "connect to several addresses" feature.

A CurveCP client tries a Hello packet to the first address, then a Hello packet to the second address, etc. If one of the servers is up then the connection will succeed promptly. This drastically reduces the impact of a single-server outage: the outage typically wastes only 1 packet and only about 1 second with CurveCP, rather than 5 packets and 180 seconds with TCP.

Load-balancing DNS servers reduce the impact of outages in a different way, by quickly removing the advertisements for servers that are down. However, load-balancing DNS does not prevent failures for users who received an advertisement of a server just before that server crashed, while CurveCP does prevent these failures. Note that neither approach prevents failures when a server crashes in the middle of a connection.

30.4 Mobile clients

Bob has an open SSH connection from his laptop in his hotel room. He closes the laptop, checks out of the hotel, walks down the street to an Internet cafe, opens the laptop, and acquires a new IP address. All his open SSH connections are now unresponsive, because the server is sending responses to the laptop's old IP address. Bob has to close the SSH connections and make new connections.

CurveCP supports mobile clients. If the server sees a new address attached to a verified packet from the client, the server tries sending its response to that address. If the client acknowledges that response from the same new address, the server switches to that address for subsequent packets, smoothly migrating the connection to that address.

The client does not cryptographically authenticate (and in general does not have any secure way to know) its own address. If an eavesdropping attacker takes a packet from the client, forges a new packet with the same contents but the attacker's address, and prevents the original packet from reaching the server (for example, by flooding the network), then the server will try sending its response to the attacker's address. If the attacker then forwards that response to the client and similarly modifies the address on the client's acknowledgment, then the server will send subsequent packets to the attacker's address. However, the client will continue sending packets from its original address, and the only way for the attacker to continue misdirecting the server's packets is to continue blocking the client's packets. An attacker with this much power can successfully deny service, but could also have denied service without any mobility features. Note that the attacker still cannot compromise the confidentiality and integrity of the connection.

Chapter 31

The CurveCP packet format

This chapter describes the four different types of packets that appear in CurveCP: Hello, Cookie, Initiate, and Message.

31.1 Connection overview

A CurveCP connection begins with a Hello packet from the client, a Cookie packet from the server, and an Initiate packet from the client. The server is free to send any number of Message packets after it sees the Initiate packet. The client is free to send any number of Message packets after it sees the server's first Message packet.

If the client does not see a Cookie packet then it will send another Hello packet. The server sends a Cookie packet in response to each Hello packet, rather than limiting the client to one Hello packet. Similarly, the client can send several Initiate packets.

The following diagram indicates how these packets are encrypted and summarizes the contents of the packets. Important notation: Box[X](C→S) is a cryptographic box, encrypting and authenticating X from the client's public key C to the server's public key S. The only people who can create or decipher Box[X](C→S) are the people who know the secret key corresponding to C and the people who know the secret key corresponding to S. (Note to readers familiar with BAN logic etc.: Box[X](C→S) should not be confused with the traditional concept $\{X\}_K$ of something encrypted under secret key K without authentication.)

Client to server	Server to client
Hello packet: (C',0,Box[0'](C'→S)) where C' is the client's short-term public key and S is the server's long-term public key and 0 is zero-padding and 0' is zero-padding	
	Cookie packet: (Box[S',K](S→C')) where S' is the server's short-term public key and K is a cookie
Initiate packet with Vouch subpacket: (C',K,Box[C,V,N,...](C'→S')) where C is the client's long-term public key and V=Box[C'](C→S) and N is the server's domain name and ... is a message	
	Message packet: (Box[...](S'→C')) where ... is a message
Message packet: (C',Box[...](C'→S')) where ... is a message	
	Message packet: (Box[...](S'→C')) where ... is a message
Message packet: (C',Box[...](C'→S')) where ... is a message	

The cookie K is Box[C',s'](t), where s' is the secret key corresponding to S', and t is a secret "minute key" maintained by the server. This is a cryptographic box that can be created and understood only by t.

31.2 Summary of the packet format

Each packet begins with an 8-byte identifier of the type of packet. If the packet happens to be delivered to some non-CurveCP application then these bytes will prevent the non-CurveCP application from being confused. A secondary function of this identifier is to allow, e.g., the server to quickly distinguish Hello packets from Initiate packets.

Each packet continues with a 16-byte receiver extension and a 16-byte sender extension. Gateways can process these extensions without parsing the rest of the packet.

Each packet from the client continues with the client's short-term public key C'. The server reuses this key as an identifier to index active connections.

Each packet continues with a cryptographic box (preceded by padding for Hello packets, and preceded by the server's cookie K for Initiate packets), as summarized in the diagram above. A nonce is attached to each cryptographic box, preventing replays and preventing confusion between the boxes in different types of packets; the nonces are not shown in the diagram above. A packet with an unopenable box (i.e., a box that fails cryptographic verification) is discarded by the receiver.

31.3 Client Hello packet details

A Hello packet is a 224-byte packet with the following contents:

- 8 bytes: the ASCII bytes “QvnQ5XlH”.
- 16 bytes: the server’s extension.
- 16 bytes: the client’s extension.
- 32 bytes: the client’s short-term public key C’.
- 64 bytes: all zero.
- 8 bytes: a client-selected compressed nonce in little-endian form. This compressed nonce is implicitly prefixed by “CurveCP-client-H” to form a 24-byte nonce.
- 80 bytes: a cryptographic box encrypted and authenticated to the server’s long-term public key S from the client’s short-term public key C’ using this 24-byte nonce. The 64-byte plaintext inside the box has the following contents:
  - 64 bytes: all zero.

Current servers enforce the 64-byte length requirement but *do not* enforce the all-zero requirement. The all-zero bytes in this packet are an anti-amplification mechanism, ensuring that Hello packets are as long as Cookie packets; this is why the 64-byte length requirement is enforced. They are also an extension mechanism, allowing future protocol extensions such as hashcash; this is why the all-zero requirement is not enforced. Clients must nevertheless be careful to follow the all-zero requirement to avoid confusing future servers that support extensions.

31.4 Server Cookie packet details

A Cookie packet is a 200-byte packet with the following format:

- 8 bytes: the ASCII bytes “RL3aNMXK”.
- 16 bytes: the client’s extension.
- 16 bytes: the server’s extension.
- 16 bytes: a server-selected compressed nonce in little-endian form. This compressed nonce is implicitly prefixed by “CurveCPK” to form a 24-byte nonce.
- 144 bytes: a cryptographic box encrypted and authenticated to the client’s short-term public key C’ from the server’s long-term public key S using this 24-byte nonce. The 128-byte plaintext inside the box has the following contents:

  - 32 bytes: the server's short-term public key S'.
  - 96 bytes: the server's cookie.

31.5 Client Initiate packet details

An Initiate packet is a (544+M)-byte packet with the following contents, where M is a multiple of 16 between 16 and 640:

- 8 bytes: the ASCII bytes "QvnQ5XlI".
- 16 bytes: the server's extension.
- 16 bytes: the client's extension.
- 32 bytes: the client's short-term public key C'.
- 96 bytes: the server's cookie.
- 8 bytes: a client-selected compressed nonce in little-endian form. This compressed nonce is implicitly prefixed by "CurveCP-client-I" to form a 24-byte nonce.
- 368+M bytes: a cryptographic box encrypted and authenticated to the server's short-term public key S' from the client's short-term public key C' using this 24-byte nonce. The (352+M)-byte plaintext inside the box has the following contents:
  - 32 bytes: the client's long-term public key C.
  - 16 bytes: a client-selected compressed nonce in little-endian form. This compressed nonce is implicitly prefixed by "CurveCPV" to form a 24-byte nonce.
  - 48 bytes: a cryptographic box encrypted and authenticated to the server's long-term public key S from the client's long-term public key C using this 24-byte nonce. The 32-byte plaintext inside the box has the following contents:
    - 32 bytes: the client's short-term public key C'.
  - 256 bytes: the server's domain name, in DNS format (between 1 and 255 bytes), zero-padded to 256 bytes.
  - M bytes: a message.

31.6 Server Message packet details

A Message packet from the server is a (64+M)-byte packet with the following contents, where M is a multiple of 16 between 16 and 1088:

- 8 bytes: the ASCII bytes "RL3aNMXM".
- 16 bytes: the client's extension.
- 16 bytes: the server's extension.
- 8 bytes: a server-selected compressed nonce in little-endian form. This compressed nonce is implicitly prefixed by "CurveCP-server-M" to form a 24-byte nonce.

- 16+M bytes: a cryptographic box encrypted and authenticated to the client's short-term public key C' from the server's short-term public key S' using this 24-byte nonce. The M-byte plaintext inside the box has the following contents:
  - M bytes: a message.

31.7 Client Message packet details

A Message packet from the client is a (96+M)-byte packet with the following contents, where M is a multiple of 16 between 16 and 1088:

- 8 bytes: the ASCII bytes "QvnQ5XlM".
- 16 bytes: the server's extension.
- 16 bytes: the client's extension.
- 32 bytes: the client's short-term public key C'.
- 8 bytes: a client-selected compressed nonce in little-endian form. This compressed nonce is implicitly prefixed by "CurveCP-client-M" to form a 24-byte nonce.
- 16+M bytes: a cryptographic box encrypted and authenticated to the server's short-term public key S' from the client's short-term public key C' using this 24-byte nonce. The M-byte plaintext inside the box has the following contents:
  - M bytes: a message.

Chapter 32

Messages in CurveCP

This chapter explains the format of CurveCP messages.

CurveCP messages should not be confused with CurveCP packets. Packets are copied to and from the network and are visible to eavesdroppers. Some packets contain encrypted authenticated messages; the contents of the messages are not visible to eavesdroppers.

CurveCP messages should also not be confused with the streams of data sent by applications that use CurveCP: a byte stream from the client to the server, and a byte stream from the server to the client. Messages contain blocks of data from these streams, but they also contain extra header information, for example to accommodate lost packets.

A byte stream is a string of bytes, between 0 bytes and $2^{60} - 1$ bytes (allowing more than 200 gigabits per second continuously for a year), followed by either success or failure. The bytes in an N-byte stream have positions 0,1,2,...,N-1; the success/failure bit has position N. A message from the sender can include a block of bytes of data from anywhere in the stream; a block can include as many as 1024 bytes. A message from the receiver can acknowledge one or more ranges of data that have been successfully received. The first range acknowledged in a message always begins with position 0. Subsequent ranges acknowledged in the same message are limited to 65535 bytes. Each range boundary sent by the receiver matches a boundary of a block previously sent by the sender, but a range normally includes many blocks.

Once the receiver has acknowledged a range of bytes, the receiver is taking responsibility for all of those bytes; the sender is expected to discard those bytes and never send them again. (This differs from the behavior of TCP SACK: TCP allows selective acknowledgments to be subsequently retracted.) The sender *can* send the bytes again; usually this occurs because the first acknowledgment was lost. The receiver discards the redundant bytes and generates a new acknowledgment covering those bytes.

32.1 What are the contents of a message?

A message has the following contents:

- 4 bytes: a message ID chosen by the sender.
- 4 bytes: if nonzero, a message ID received by the sender immediately before this message was sent.
- 8 bytes: a 64-bit unsigned integer in little-endian form, the number of bytes in the first

range being acknowledged as part of this message. A range can include 0 bytes, in which case it does not actually acknowledge anything.

- 4 bytes: a 32-bit unsigned integer in little-endian form, the number of bytes between the first range and the second range.
- 2 bytes: a 16-bit unsigned integer in little-endian form, the number of bytes in the second range.
- 2 bytes: a 16-bit unsigned integer in little-endian form, the number of bytes between the second range and the third range.
- 2 bytes: a 16-bit unsigned integer in little-endian form, the number of bytes in the third range.
- 2 bytes: a 16-bit unsigned integer in little-endian form, the number of bytes between the third range and the fourth range.
- 2 bytes: a 16-bit unsigned integer in little-endian form, the number of bytes in the fourth range.
- 2 bytes: a 16-bit unsigned integer in little-endian form, the number of bytes between the fourth range and the fifth range.
- 2 bytes: a 16-bit unsigned integer in little-endian form, the number of bytes in the fifth range.
- 2 bytes: a 16-bit unsigned integer in little-endian form, the number of bytes between the fifth range and the sixth range.
- 2 bytes: a 16-bit unsigned integer in little-endian form, the number of bytes in the sixth range.
- 2 bytes: a 16-bit unsigned integer in little-endian form, the sum of the following integers:
  - D, an integer between 0 and 1024, the size of the data block being sent as part of this message.
  - SUCC, either 0 or 2048, where 2048 means that this block is known to be at the end of the stream followed by success.
  - FAIL, either 0 or 4096, where 4096 means that this block is known to be at the end of the stream followed by failure.
- 8 bytes: a 64-bit unsigned integer in little-endian form, the position of the first byte in the data block being sent. If D=0 but SUCC>0 or FAIL>0 then this is the success/failure position, i.e., the total number of bytes in the stream.
- Zero-padding. This padding produces a total message length that is a multiple of 16 bytes, at least 16 bytes and at most 1088 bytes.
- D bytes: the data block being sent.

The zero-padding is not limited to 15 bytes; i.e., it does not necessarily produce the smallest possible multiple of 16 bytes. Some message creators pad to a more restricted set of lengths to reduce the amount of information that the message length reveals about the length of the data block. Note, however, that traffic analysis continues to reveal a tremendous amount of information.

32.2 Does CurveCP provide flow control?

Yes. Decongestion, when used properly, automatically provides flow control. The receiver simply adds the sender's incoming messages to a queue, without looking at the messages. The receiver can impose whatever size limits it wants upon the queue, static or dynamic, without notification to the sender. When the receiver is ready to handle more data, it processes the message at the front of the queue, generating an acknowledgment if necessary.

Using decongestion to handle flow control simplifies the protocol, eliminating any need to coordinate window updates and eliminating common mistakes such as silly-window syndrome. It slightly reduces packet sizes and sometimes eliminates entire window-update packets. It can slightly increase traffic on the occasions that a computer is unable to keep up with its own network link, but decongestion automatically keeps this increase under control.

32.3 Does CurveCP allow configuration of the maximum segment size?

No. The maximum CurveCP data-block size is 1024 bytes. The maximum CurveCP message size is 1088 bytes (or 640 bytes inside Initiate packets). The maximum CurveCP packet size, including lower-layer overhead, is smaller than the 1280-byte datagrams allowed by IPv6 on all networks, the 1492-byte datagrams that have always been supported by Ethernet, the 2272-byte datagrams supported by 802.11 wireless networks, etc.

CurveCP's total packet overhead, including lower-layer overhead, is under 15% for bulk data transfer. In theory, allowing larger packets would measurably reduce this overhead; in practice, attempts to maximize packet size have a long and continuing history of damaging Internet connectivity.

Chapter 33

Message-handling programs

A traditional UNIX-style server such as `ftpd` handles just one network connection, reading input from stdin and writing output to stdout. A “superserver” such as `inetd` or `tcpserver` listens for network connections and starts a separate server process for each connection.

The CurveCP command-line tools have an extra level of modularity. The `curvecpserver` superserver listens for network connections. For each connection, `curvecpserver` starts the `curvecpmessage` message handler; `curvecpmessage` then starts a server such as `ftpd`. Then `ftpd` sends a stream of data to `curvecpmessage`, which in turn sends messages to `curvecpserver`, which encrypts and authenticates the messages and sends them inside network packets. At the same time `curvecpclient` receives network packets, verifies and decrypts messages inside the packets, and passes the messages to `curvecpmessage`; `curvecpmessage` sends a stream of data to `ftpd`. The same `curvecpmessage` tool is also used by `curvecpclient`.

`curvecpserver` and `curvecpclient` can use programs other than `curvecpmessage`. Those programs can directly generate messages in the CurveCP message format without talking to separate tools such as `ftpd`; or they can support a completely different protocol that reuses CurveCP’s cryptographic layer but transmits different kinds of messages.

This chapter explains what programmers have to do to write `curvecpmessage` replacements that talk to `curvecpserver` and `curvecpclient`.

33.1 Incoming messages

File descriptor 8 is a pipe. Read from this pipe a length byte n, between 1 and 68, and a 16*n-byte message. Repeat. The pipe is set to non-blocking mode; be prepared for EAGAIN and EWOULDBLOCK, even in the middle of a message.

This pipe reading must always be active. The `curvecpclient` and `curvecpserver` programs assume that every message is read immediately. If you can’t handle a message immediately, read it and put it onto a queue. If you don’t have queue space, throw the message away; this shouldn’t cause trouble, since you have to be able to handle missing messages in any case.

33.2 Outgoing messages

File descriptor 9 is a pipe. Write to this pipe a length byte *n*, between 1 and 68, and a 16**n*-byte message. Repeat. The pipe is set to non-blocking mode; be prepared for EAGAIN and EWOULDBLOCK, even in the middle of a message.

As a client, do not use length bytes above 40 until a message has arrived from the server. (The messages inside CurveCP Initiate packets are limited to 640 bytes.)

The CurveCP server does not start until it has received a message from the client. Furthermore, the CurveCP server must receive this message within 60 seconds of the client starting up. (The CurveCP Initiate packet is valid for only 60 seconds after the corresponding CurveCP Cookie packet.) This does not mean that the client must start sending messages immediately, but it does mean that waiting for more than a second to send a message is a bad idea.

Chapter 34

The CurveCP command-line tools

This chapter describes the five command-line CurveCP tools provided by NaCl.

`curvecpmakekey` creates a public key and a corresponding secret key. It has one argument, the name of a directory where the keys will be stored.

`curvecpprintkey` prints a public key as 64 hexadecimal digits. It has one argument, the name of the directory created by `curvecpmakekey`.

`curvecpmessage` is a generic message-handling program (see the previous chapter) that runs existing command-line clients and servers. This program handles conversion of data into messages and vice versa; retransmission of messages; and congestion control. The arguments to `curvecpmessage` are `-c` followed by the name of the passive client to run, `-C` followed by the name of the active client to run, or `-s` followed by the name of the server to run. The difference between passive clients and active clients is that active clients send data first while passive clients wait for the server to send data first. In the passive case, `curvecpmessage` sends a blank message immediately (triggering an Initiate packet that starts the server); in the active case, `curvecpmessage` waits for a second for data from the client.

`curvecpclient` is analogous to `tcpclient`. It makes a CurveCP connection to a specified server and runs a specified program. Its arguments are the server's DNS name, the server's public key (in hexadecimal), the server's IP address, the server's UDP port, the server's extension (in hexadecimal), and the program to run. Typically the program to run is `curvecpmessage -c ...`; for example, `curvecpmessage -c multitee 0:7 6:1e0` has a similar effect to `netcat`.

`curvecpserver` is a superserver analogous to `inetd` or `tcpserver`. It listens for CurveCP connections and, for each connection, starts a specified program. Its arguments are the server's DNS name, the name of the key directory on disk, the server's IP address (a local address on this machine), the server's UDP port, the server's extension (in hexadecimal), and the program to run. Typically the program to run is `curvecpmessage -s ...`, where the `...` is an existing program such as `smtpd` or `httpd`.

Bibliography

[1] — (no editor), *20th annual symposium on foundations of computer science*, IEEE Computer Society, New York, 1979. MR 82a:68004. See [36].

[2] Jee Hea An, *Authenticated encryption in the public-key setting: security notions and analyses* (2001). URL: `http://eprint.iacr.org/2001/079`. Citations in this document: §6.4.

[3] Jean-Philippe Aumasson, Simon Fischer, Shahram Khazaei, Willi Meier, Christian Rechberger, *New features of Latin dances: analysis of Salsa, ChaCha, and Rumba* (2007); see also newer version [4]. URL: `http://eprint.iacr.org/2007/472`.

[4] Jean-Philippe Aumasson, Simon Fischer, Shahram Khazaei, Willi Meier, Christian Rechberger, *New features of Latin dances: analysis of Salsa, ChaCha, and Rumba*, in [29] (2007), 470–488; see also older version [3]. Citations in this document: §21.

[5] Rana Barua, Tanja Lange (editors), *Progress in cryptology—INDOCRYPT 2006, 7th international conference on cryptology in India, Kolkata, India, December 11–13, 2006, proceedings*, Lecture Notes in Computer Science, 4329, Springer, 2006. ISBN 3-540-49767-6. See [23].

[6] Mihir Bellare, Joe Kilian, Phillip Rogaway, *The security of cipher block chaining*, in [21] (1994), 341–358; see also newer version [7].

[7] Mihir Bellare, Joe Kilian, Phillip Rogaway, *The security of the cipher block chaining message authentication code*, Journal of Computer and System Sciences **61** (2000), 362–399; see also older version [6]. ISSN 0022–0000. URL: `http://www-cse.ucsd.edu/~mihir/papers/cbc.html`. Citations in this document: §9.3, §10.3.

[8] Mihir Bellare, Chanathip Namprempre, *Authenticated encryption: relations among notions and analysis of the generic composition paradigm*, in [31] (2000), 531–545. URL: `http://www-cse.ucsd.edu/~mihir/papers/oem.html`. Citations in this document: §8.3.

[9] Daniel J. Bernstein, *Crypto flaw in secure mail standards (*`sci.crypt` *posting)* (2001). URL: `http://groups.google.com/group/sci.crypt/msg/ec5c18b23b11d82c`. Citations in this document: §6.4.

[10] Daniel J. Bernstein, *Floating-point arithmetic and message authentication* (2004). URL: `http://cr.yp.to/hash127.html`. ID `dabadd3095644704c5cbe9690ea3738e`. Citations in this document: §23.

[11] Daniel J. Bernstein, *The Poly1305-AES message-authentication code*, in [24] (2005), 32–49. URL: `http://cr.yp.to/papers.html#poly1305`. ID `0018d9551b5546d97c340e0dd8cb5750`. Citations in this document: §23, §23, §23.

[12] Daniel J. Bernstein, *Salsa20 specification* (2005). URL: `http://cr.yp.to/snuffle.html`. Citations in this document: §21, §22, §22.

[13] Daniel J. Bernstein, *Curve25519: new Diffie-Hellman speed records*, in [39] (2006), 207–228. URL: `http://cr.yp.to/papers.html#curve25519`. ID `4230efdfa673480fc079449d90f322c0`. Citations in this document: §7.6, §16, §16, §16, §16, §16, §19.

[14] Daniel J. Bernstein, *Convenient x-coordinate-only EC signing (*`sci.crypt` *posting)*. URL: `http://www.derkeiler.com/Newsgroups/sci.crypt/2006-09/msg00642.html`. Citations in this document: §25.

[15] Daniel J. Bernstein, *Polynomial evaluation and message authentication* (2007). URL: `http://cr.yp.to/papers.html#pema`. ID `b1ef3f2d385a926123e1517392e20f8c`. Citations in this document: §23.

[16] Daniel J. Bernstein, *Extending the Salsa20 nonce* (2008). URL: `http://cr.yp.to/papers.html#xsalsa`. ID `c4b172305ff16e1429a48d9434d50e8a`. Citations in this document: §21, §21.

[17] Daniel J. Bernstein, *The dnssecamp tool* (2011). URL: `http://dnscurve.org/dnssecamp.html`. Citations in this document: §28.5.

[18] Daniel J. Bernstein, Tanja Lange (editors), *eBACS: ECRYPT Benchmarking of Cryptographic Systems* (2011). URL: `http://bench.cr.yp.to`. Citations in this document: §1.1.

[19] John Black, Martin Cochran, *MAC reforgeability*, in [22] (2009), 345–362. URL: `http://eprint.iacr.org/2006/095`. Citations in this document: §23.

[20] Paul Crowley, *Truncated differential cryptanalysis of five rounds of Salsa20*, in Workshop Record of SASC 2006: Stream Ciphers Revisted, eSTREAM technical report 2005/073 (2005). URL: `http://www.ecrypt.eu.org/stream/papers.html`. Citations in this document: §21.

[21] Yvo Desmedt (editor), *Advances in cryptology—CRYPTO '94*, Lecture Notes in Computer Science, 839, Springer, Berlin, 1994. See [6].

[22] Orr Dunkelman (editor), *Fast software encryption, 16th international workshop, FSE 2009, Leuven, Belgium, February 22–25, 2009, revised selected papers*, Lecture Notes in Computer Science, 5665, Springer, 2009. ISBN 978-3-642-03316-2. See [19].

[23] Simon Fischer, Willi Meier, Côme Berbain, Jean-François Biasse, Matthew J. B. Robshaw, *Non-randomness in eSTREAM candidates Salsa20 and TSC-4*, in [5] (2006), 2–16. Citations in this document: §21.

[24] Henri Gilbert, Helena Handschuh (editors), *Fast software encryption: 12th international workshop, FSE 2005, Paris, France, February 21–23, 2005, revised selected papers*, Lecture Notes in Computer Science, 3557, Springer, 2005. ISBN 3–540–26541–4. See [11].

[25] Helena Handschuh, Bart Preneel, *Key-recovery attacks on universal hash function based MAC algorithms*, in [35] (2008), 144–161. Citations in this document: §23.

[26] Don Johnson, Alfred Menezes, Scott A. Vanstone, *The elliptic curve digital signature algorithm (ECDSA)*, International Journal of Information Security **1** (2001), 36–63. Citations in this document: §25.

[27] David A. McGrew, Scott R. Fluhrer, *Multiple forgery attacks against Message Authentication Codes* (2005). URL: `http://eprint.iacr.org/2005/161`. Citations in this document: §23.

[28] Victor S. Miller, *Use of elliptic curves in cryptography*, in [38] (1986), 417–426. MR 88b:68040. Citations in this document: §16.

[29] Kaisa Nyberg (editor), *Fast software encryption, 15th international workshop, FSE 2008, Lausanne, Switzerland, February 10-13, 2008, revised selected papers* (2008). ISBN 978-3-540-71038-7. See [4].

[30] Kaisa Nyberg, Henri Gilbert, Matt Robshaw, *Galois MAC with forgery probability close to ideal* (2005). URL: `http://csrc.nist.gov/groups/ST/toolkit/BCM/documents/comments/General_Comments/papers/Nyberg_Gilbert_and_Robshaw.pdf`. Citations in this document: §23.

[31] Tatsuaki Okamoto (editor), *Advances in cryptology: ASIACRYPT 2000*, Lecture Notes in Computer Science, 1976, Springer, Berlin, 2000. ISBN 3–540–41404–5. MR 2002d:94046. See [8].

[32] William Stein (editor), *Sage Mathematics Software (Version 3.0.2)*, The Sage Group, 2008. URL: `http://www.sagemath.org`. Citations in this document: §15.

[33] Yukiyasu Tsunoo, Teruo Saito, Hiroyasu Kubo, Tomoyasu Suzaki, Hiroki Nakashima, *Differential Cryptanalysis of Salsa20/8*, in Workshop Record of SASC 2007: The State of the Art of Stream Ciphers, eSTREAM report 2007/010 (2007). URL: `http://www.ecrypt.eu.org/stream/papers.html`. Citations in this document: §21.

[34] Matthijs van Duin, *Convenient x-coordinate-only EC signing (*`sci.crypt` *posting)* (2006). URL: `http://www.derkeiler.com/Newsgroups/sci.crypt/2006-09/msg00445.html`. Citations in this document: §25.

[35] David Wagner (editor), *Advances in Cryptology—CRYPTO 2008, 28th annual international cryptology conference, Santa Barbara, CA, USA, August 17–21, 2008, proceedings*, Lecture Notes in Computer Science, 5157, Springer, 2008. ISBN 978-3-540-85173-8. See [25].

[36] Mark N. Wegman, J. Lawrence Carter, *New classes and applications of hash functions*, in [1] (1979), 175–182; see also newer version [37]. URL: `http://cr.yp.to/bib/entries.html#1979/wegman`.

[37] Mark N. Wegman, J. Lawrence Carter, *New hash functions and their use in authentication and set equality*, Journal of Computer and System Sciences **22** (1981), 265–279; see also older version [36]. ISSN 0022–0000. MR 82i:68017. URL: `http://cr.yp.to/bib/entries.html#1981/wegman`. Citations in this document: §23.

[38] Hugh C. Williams (editor), *Advances in cryptology: CRYPTO '85*, Lecture Notes in Computer Science, 218, Springer, Berlin, 1986. ISBN 3–540–16463–4. See [28].

[39] Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, Tal Malkin (editors), *9th international conference on theory and practice in public-key cryptography, New York, NY, USA, April 24–26, 2006, proceedings*, Lecture Notes in Computer Science, 3958, Springer, Berlin, 2006. ISBN 978–3–540–33851–2. See [13].