# The security impact of a new cryptographic library

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

[1] Department of Computer Science
University of Illinois at Chicago, Chicago, IL 60607–7053, USA
djb@cr.yp.to
[2] Department of Mathematics and Computer Science
Technische Universiteit Eindhoven
P.O. Box 513, 5600MB Eindhoven, the Netherlands
tanja@hyperelliptic.org
[3] Research Center for Information Technology Innovation and
Institute of Information Science
Academia Sinica
No. 128 Academia Road, Section 2, Nankang, Taipei 11529, Taiwan
peter@cryptojedi.org

**Abstract.** This paper introduces a new cryptographic library, NaCl, and explains how the design and implementation of the library avoid various types of cryptographic disasters suffered by previous cryptographic libraries such as OpenSSL.

**Keywords:** confidentiality, integrity, simplicity, speed, security

## 1 Introduction

For most cryptographic operations there exist widely accepted standards, such as the Advanced Encryption Standard (AES) for secret-key encryption and 2048-bit RSA for public-key encryption. These primitives have been extensively studied, and breaking them is considered computationally infeasible on any existing computer cluster.

For each of these cryptographic primitives there exist various implementations and software libraries, and it has become common best practice in the development of secure systems to use the implementations in these libraries as building blocks. One should thus expect that the cryptographic layer of modern

This work was supported by the National Science Foundation under grant 1018836; by the European Commission through the ICT Programme under Contract ICT-2007-216499 CACE and Contract ICT-2007-216676 ECRYPT II; and by the National Science Council, National Taiwan University and Intel Corporation under Grant NSC99-2911-I-002-001 and 99-2218-E-001-007. Part of this work was carried out when Peter Schwabe was employed by National Taiwan University; part of this work was carried out when Peter Schwabe was employed by Technische Universiteit Eindhoven. Permanent ID of this document: 5f6fc69cc5a319aecba43760c56fab04. Date: 2011.12.01.

information systems does not expose any vulnerabilities to attackers. Unfortunately this expectation is far from reality, as demonstrated by one embarrassing cryptographic failure after another.

**A new cryptographic library: NaCl.** To address the underlying problems we have designed and implemented a cryptographic library named NaCl. The name is pronounced "salt" and stands for "Networking and Cryptography Library". This paper discusses only the cryptographic part of NaCl; the networking part is still in prototype form.

NaCl is in the public domain and is available from `http://nacl.cr.yp.to` and `http://nacl.cace-project.eu`, along with extensive documentation. The signature component of NaCl is integrated only into the latest development version, which is not yet online, but the same code is available separately as part of the SUPERCOP benchmarking package at `http://bench.cr.yp.to`. NaCl steers clear of all patents that we have investigated and has not received any claims of patent infringement.

The first announcement of NaCl was in 2008. We considered changing the name of the project when Google announced Native Client, but decided that there was no real risk of confusion. The first release of NaCl was in 2009 but was missing some of the important features discussed in this paper; the C++ NaCl API was not released until 2010, for example, and signatures were not released until 2011.

A research paper on cryptographic software normally focuses on optimizing the choice and implementation of a single cryptographic primitive at a specified security level: for example, [10] reports speed records for signatures at a $2^{128}$ security level. This paper is different. Our goal is to analyze the real-world security benefits of switching from an existing cryptographic library such as OpenSSL [27] to a completely new cryptographic library. Some of these security benefits are tied to performance, as discussed later, so we naturally use the results of papers such as [10]; but what is new in this paper is the security analysis.

**Acknowledgments.** Several of the implementations in NaCl are partially or entirely from third parties. The portability of NaCl relies on the `ref` implementation of Curve25519 written by Matthew Dempsky (Mochi Media, now Google). From 2009 until 2011 the speed of NaCl on common Intel/AMD CPUs relied on the `donna` and `donna_c64` implementations of Curve25519 written by Adam Langley (Google) — which, interestingly, also appear in Apple's acknowledgments [3] for iOS 4. The newest implementations of Curve25519 and Ed25519 were joint work with Niels Duif (Technische Universiteit Eindhoven) and Bo-Yin Yang (Academia Sinica). The `core2` implementation of AES was joint work with Emilia Käsper (Katholieke Universiteit Leuven, now Google).

Prototype Python wrappers around C NaCl have been posted by Langley; by Jan Mojzis; and by Sean Lynch (Facebook). We will merge these wrappers and integrate them into the main NaCl release as a single supported Python NaCl, in the same way that we support C++ NaCl.

## 2 The NaCl API

The reader is assumed to be familiar with the fact that most Internet communication today is cryptographically unprotected. The primary goal of NaCl is to change this: to cryptographically protect every network connection, providing strong confidentiality, strong integrity, and state-of-the-art availability against attackers sniffing or modifying network packets.

Confidentiality is limited to packet contents, not packet lengths and timings, so users still need anti-traffic-analysis tools: route obfuscators such as Tor [35], timing obfuscators, etc. Of course, users also need vastly better software security in operating systems, web browsers, document viewers, etc. Cryptography is only one part of security.

This section introduces the functions provided by NaCl, with an emphasis on the simplicity of these functions: more precisely, the simplicity that these functions bring to cryptographic applications. This section is meant mostly as background for the security analysis in subsequent sections, but there are enough differences between the NaCl API and previous APIs to justify a discussion of the details.

**The `crypto_box` API.** The central job of a cryptographic library is **public-key authenticated encryption**. The general setup is that a sender, Alice, has a packet to send to a receiver, Bob. Alice scrambles the packet using Bob's public key and her own secret key. Bob unscrambles the packet using Alice's public key and his own secret key. "Encryption" refers to confidentiality: an attacker monitoring the network is unable to understand the scrambled packet. "Authenticated" refers to integrity: an attacker modifying network packets is unable to change the packet produced by Bob's unscrambling. (Availability, to the extent that it is not inherently limited by network resources, is provided by higher-level networking protocols that retransmit lost packets.)

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

- Alice generates a random AES key.
- Alice uses the AES key to encrypt the packet.
- Alice hashes the encrypted packet using SHA-256.
- Alice reads her RSA secret key from "wire format."
- Alice uses her RSA secret key to sign the hash.
- Alice reads Bob's RSA public key from wire format.
- Alice uses Bob's public key to encrypt the AES key, hash, and signature.
- Alice converts the encrypted key, hash, and signature to wire format.
- Alice concatenates with the encrypted packet.

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

NaCl gives Alice a simple high-level `crypto_box` function that does everything in one step, putting a packet into a box that is protected against espionage and sabotage:

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

The function takes the sender's secret key `sk` (32 bytes), the recipient's public key `pk` (also 32 bytes), a packet `m`, and a nonce `n` (24 bytes), and produces an authenticated ciphertext `c` (16 bytes longer than `m`). All of these objects are C++ `std::string` variables, represented in wire format as sequences of bytes suitable for transmission; the `crypto_box` function automatically handles all necessary conversions, initializations, etc. Bob's operation is just as easy, with the keys and packets reversed:

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

Each side begins with

```
pk = crypto_box_keypair(&sk)
```

to generate a secret key and a public key in the first place.

The C NaCl API has the same function names but more arguments: for example, `std::string m` is replaced by `unsigned char *m` and `unsigned long long mlen`. Failures are indicated by exceptions in C++ NaCl and a `-1` return value in C NaCl.

**Validation of the API.** The API described above might seem *too* simple to support the needs of real-world applications. We emphasize that NaCl has already been integrated into high-security applications that are running on the Internet today.

DNSCurve [8], designed by the first author, provides high-security authenticated encryption for DNS queries between a DNS resolver and a DNS server. (The server's public key is provided by its parent DNS server, which of course also needs to be secured; the client's public key is provided as part of the protocol.) NaCl has been used successfully for several independent DNSCurve implementations, including an implementation used [17] by the OpenDNS resolvers, which handle billions of DNS queries a day from millions of computers and automatically use DNSCurve for any DNSCurve server. Other applications of NaCl so far include the QuickTun VPN software [30]; the Ethos operating system [32]; and the first author's prototype implementation of CurveCP [9], a high-security cryptographic version of TCP.

C NaCl allows `crypto_box` to be split into two steps, `crypto_box_beforenm` and `crypto_box_afternm`, slightly compromising simplicity but gaining extra speed as discussed in Section 4. The `beforenm` step preprocesses `pk` and `sk`, preparing to handle any number of messages; the `afternm` step handles `n` and `m`. Most applications actually use this two-step procedure.

**Nonces.** The `crypto_box` API leaves nonce generation to the caller. This is not meant to suggest that nonce generation is not part of the cryptographer's job; on the contrary, we believe that cryptographers should take responsibility not just for nonces but also for other security aspects of high-level network protocols. The exposure of nonces simply reflects the fact that nonces are integrated into high-level protocols in different ways.

It might seem simplest to always generate a random 24-byte nonce `n`, and to transmit this nonce as part of the authenticated ciphertext; 24-byte random strings have negligible chance of colliding. If ciphertexts are long then one can tolerate the costs of generating this randomness and of expanding each ciphertext by 24 bytes. However, random nonces do nothing to stop the simplest type of forgery, namely a replay. One standard strategy to prevent replays is to include an increasing number in each packet and to reject any packet whose number is not larger than the number in the last verified packet; using these sequence numbers as nonces is simpler than giving each packet a number *and* a random nonce. On the other hand, choosing public nonces as sequence numbers means giving away traffic information that would otherwise be somewhat more expensive for an attacker to collect. Several different solutions appear in the literature; constraints on nonce generation are often tied directly to questions of the security that users expect.

Current applications of NaCl, such as DNSCurve and CurveCP, have different requirements regarding nonces, replays, forward secrecy, and many other security issues at a higher level than the `crypto_box` API. A nonceless API would require higher-level complications in all of these applications, and would not simplify their security analysis.

**The `crypto_sign` API.** Sometimes confidentiality is irrelevant: Alice is sending a public message to many people. In this situation it is helpful for a cryptographic library to provide **public-key signatures**: Alice scrambles the message using her own secret key, and Bob unscrambles the message using Alice's public key. Alice's operations are independent of Bob, allowing the scrambled message to be broadcast to any number of receivers. Signatures also provide non-repudiation, while authenticators are always repudiable.

NaCl provides simple high-level functions for signatures: Alice uses

```
pk = crypto_sign_keypair(&sk)
```

to generate a key pair (again 32 bytes for the public key but 64 bytes for the secret key), and

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

to create a signed message (64 bytes longer than the original message). Bob uses

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

to unscramble the signed message, recovering the original message.

**Comparison to previous work.** NaCl is certainly not the first cryptographic library to promise a simple high-level API. For example, Gutmann's cryptlib library [20] advertises a "high-level interface" that "provides anyone with the ability to add strong security capabilities to an application in as little as half an hour, without needing to know any of the low-level details that make the encryption or authentication work." See [21, page 1].

There are, however, many differences between high-level APIs, as illustrated by the following example. The first code segment in the cryptlib manual [21, page 13] ("the best way to illustrate what cryptlib can do") contains the following six function calls, together with various comments:

```
cryptCreateEnvelope( &cryptEnvelope, cryptUser,
   CRYPT_FORMAT_SMIME );
cryptSetAttributeString( cryptEnvelope,
   CRYPT_ENVINFO_RECIPIENT,
   recipientName, recipientNameLength );
cryptPushData( cryptEnvelope, message, messageSize,
   &bytesIn );
cryptFlushData( cryptEnvelope );
cryptPopData( cryptEnvelope, encryptedMessage, encryptedSize,
   &bytesOut );
cryptDestroyEnvelope( cryptEnvelope );
```

This sequence has a similar effect to NaCl's

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

where `message` is the plaintext `m` and `encryptedMessage` is the ciphertext `c`.

The most obvious difference between these examples is in conciseness: cryptlib has separate functions

- `cryptCreateEnvelope` to allocate storage,
- `cryptSetAttributeString` to specify the recipient,
- `cryptPushData` to start the plaintext input,
- `cryptFlushData` to finish the plaintext input,
- `cryptPopData` to extract the ciphertext, and
- `cryptDestroyEnvelope` to free storage,

while NaCl handles everything in one function. The cryptlib program must also call `cryptInit` at some point before this sequence.

A much less obvious difference is in reliability. For example, if the program runs out of memory, NaCl will raise an exception, while the above cryptlib code will fail in unspecified ways, perhaps silently corrupting or leaking data. The cryptlib manual [21, page 35] states that the programmer is required to check that each function returns `CRYPT_OK`, and that the wrong code shown above is included in the manual "for clarity". Furthermore, [21, page 53] says that if messages are large then "only some of the data may be copied in" by `cryptPushData`; the programmer is required to check `bytesIn` and loop appropriately. Trouble can occur even if messages are short and memory is ample: for example, [21, page 14] indicates that recipient public keys are retrieved from an on-disk database, but does not discuss what happens if the disk fails or if an attacker consumes all available file descriptors.

Some of the differences between these code snippets are really differences between C and C++: specifically, NaCl benefits from C++ exceptions and C++

strings, while cryptlib does not use these C++ features. For applications written in C, rather than C++, the cryptlib API should instead be compared to the C NaCl API:

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

This C NaCl function cannot raise C++ exceptions, but it also does not need to: its only possible return value is 0, indicating successful authenticated encryption. C NaCl is intended to be usable in operating-system kernels, critical servers, and other environments that cannot guarantee the availability of large amounts of heap storage but that nevertheless rely on their cryptographic computations to continue working. In particular, 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, so that stack space can be allocated in advance and guaranteed to be adequate.

Perhaps the most important difference between these NaCl and cryptlib examples is that the `crypto_box` output is authenticated and encrypted using keys from Alice and Bob, while the cryptlib output is merely encrypted to Bob without any authentication; cryptlib supports signatures but does not add them without extra programming work. There is a long history of programs omitting cryptographic authentication, incorrectly treating all successfully decrypted data as authentic, and being exploited as a result; with cryptlib, writing such programs is easier than writing programs that include proper authentication. With NaCl, high-security authenticated encryption is the easiest operation.

## 3 Security

This section presents various case studies of cryptographic disasters, and explains the features of NaCl that eliminate these types of disasters.

Two specific types of disasters are addressed in subsequent sections: Section 4 discusses users deliberately weakening or disabling cryptography to address cryptographic performance problems; Section 5 discusses cryptographic primitives being broken.

**No data flow from secrets to load addresses.** In 2005, Osvik, Shamir, and Tromer described a timing attack that discovered the AES key of the `dm-crypt` hard-disk encryption in Linux in just 65 milliseconds. See [28] and [36]. The attack process runs on the same machine but does not need any privileges (for example, it can run inside a virtual machine) and does not exploit any kernel software security holes.

This attack is possible because almost all implementations of AES, including the Linux kernel implementation, use fast lookup tables as recommended in the initial AES proposal; see [16, Section 5.2]. The secret AES key inside the kernel influences the table-load addresses, which in turn influence the state of the CPU cache, which in turn influences measurable timings of the attack process; the attack process computes the AES key from this leaked information.

NaCl avoids this type of disaster by systematically avoiding all loads from addresses that depend on secret data. All of the implementations are thus inherently protected against cache-timing attacks. This puts constraints on the implementation strategies used throughout NaCl, and also influences the choice of cryptographic algorithms in NaCl, as discussed in Section 5.

For comparison, Gutmann's cryptlib manual [21, pages 63–64] claims that cache-timing attacks (specifically "observing memory access latencies for cached vs. un-cached data") and branch-timing attacks (see below) provide almost the same "level of access" as "an in-circuit emulator (ICE)" and that there are therefore "no truly effective defences against this level of threat". We disagree. Software side channels on common CPUs include memory addresses and branch conditions but do not include, e.g., the inputs and outputs to a XOR operation; it is well known that the safe operations are adequate in theory to perform cryptographic computations, and NaCl demonstrates that the operations are also adequate in practice. Typical cryptographic code uses unsafe operations, and cache-timing attacks have been repeatedly demonstrated to be effective against such code, but NaCl's approach makes these attacks completely ineffective.

OpenSSL has responded to cache-timing attacks in a different way, not prohibiting secret load addresses but instead using complicated countermeasures intended to obscure the influence of load addresses upon the cache state. This obviously cannot provide the same level of confidence as the NaCl approach: a straightforward code review can convincingly verify the predictability of all load addresses in NaCl, while there is no similarly systematic way to verify the efficacy of other countermeasures. The review of load addresses and branch conditions (see below) has already been formalized and automated for large parts of NaCl; see [2] (which comments that "NaCl code follows strict coding policies that make it *formal verification-friendly*") and [25].

**No data flow from secrets to branch conditions.** This year Brumley and Tuveri announced that they had used a remote timing attack to find the ECDSA private key used for server authentication in a TLS handshake. See [13]. The implementation targeted in this attack is the ECDSA implementation in OpenSSL.

The underlying problem is that most scalar-multiplication (and exponentiation) algorithms involve data flow from secret data into branch conditions: i.e., certain operations are carried out if and only if the key has certain properties. In particular, the OpenSSL implementation of ECDSA uses one of these algorithms. Secret data inside OpenSSL influences the state of the CPU branch unit, which in turn influences the amount of time used by OpenSSL, which in turn influences measurable timings of network packets; the attacker computes the ECDSA key from this leaked information.

NaCl avoids this type of disaster by systematically avoiding all branch conditions that depend on secret data. This is analogous to the prohibition on secret load addresses discussed above; it has pervasive effects on NaCl's implementation strategies and interacts with the cryptographic choices discussed in Section 5.

**No padding oracles.** In 1998 Bleichenbacher successfully decrypted an RSA-encrypted SSL ciphertext by sending roughly one million variants of the cipher-

text to the server and observing the server's responses. The server would apply RSA decryption to each variant and publicly reject the (many) variants not having "PKCS #1" format. Subsequent integrity checks in SSL would defend against forgeries and reject the remaining variants, but the pattern of initial rejections already leaked so much information that Bleichenbacher was able to compute the plaintext. See [12].

NaCl has several layers of defense against this type of disaster:

- NaCl's authenticated-encryption mechanism is designed as a secure unit, always wrapping encryption inside authentication. Nothing is decrypted unless it first survives authentication, and the authenticator's entire job is to prevent the attacker from forging messages that survive authentication.
- Forged messages always follow the same path through authenticator verification, using constant time (depending only on the message length, which is public) and then rejecting the message, with no output other than the fact that the message is forged.
- Even if the attacker forges a variant of a message by sheer luck, the forgery will be visible only through the receiver accepting the message, and standard nonce-handling mechanisms in higher-level protocols will instantly reject any further messages under the same nonce. NaCl derives new authentication and encryption keys for each nonce, so the attacker will have no opportunity to study the effect of those keys on any further messages.

Note that the third defense imposes a key-agility requirement on the underlying cryptographic algorithms.

Most cryptographic libraries responded to Bleichenbacher's attack by trying to hide different types of message rejection, along the lines of the second defense; for example, [22] shows that this approach was adopted by the GnuTLS library in 2006. However, typical libraries continue to show small timing variations, so this defense by itself is not as confidence-inspiring as using strong authentication to shield decryption. Conceptually similar attacks have continued to plague cryptographic software, as illustrated by the SSH attack in [1] two years ago.

**Centralizing randomness.** In 2006 a Debian developer removed a critical line of randomness-generation code from the OpenSSL package shipped with Debian GNU/Linux. Code-verification tools had complained that the line was producing unpredictable results, and the developer did not see why the line was necessary. Until this bug was discovered in 2008 (see [31]), OpenSSL keys generated under Debian and Ubuntu were chosen from a set of size only 32768. Breaking the encryption or authentication of any communication secured with such a key was a matter of seconds.

NaCl avoids this type of disaster by simply reading bytes from the operating-system kernel's cryptographic random-number generator. Of course, the relevant code in the kernel needs to be carefully written, but reviewing that code is a much more tractable task than reviewing all of the separate lines of randomness-generation code in libraries that decide to do the job themselves. The benefits of code minimization are well understood in other areas of security; we are

constantly surprised by the amount of unnecessary complexity in cryptographic software.

A structural deficiency in the `/dev/urandom` API provided by Linux, BSD, etc. is that using it can fail, for example because the system has no available file descriptors. In this case NaCl waits and tries again. We recommend that operating systems add a reliable `urandom(x,xlen)` system call.

**Avoiding unnecessary randomness.** Badly generated random numbers were also involved in the recent collapse of the security system of Sony's PlayStation 3 gaming console. Sony used the standard elliptic-curve digital-signature algorithm, ECDSA, but ignored the ECDSA requirement of a new random secret for each message: Sony simply used a constant value for all messages. Attackers exploited this mistake to compute Sony's root signing key, as explained in [14, slides 122–130], breaking the security system of the PlayStation 3 beyond repair.

NaCl avoids this type of disaster by using *deterministic* cryptographic operations to the extent possible. The `keypair` operations use new randomness, but all of the other operations listed above produce outputs determined entirely by their inputs. Of course, this imposes a constraint upon the underlying cryptographic primitives: primitives that use randomness, such as ECDSA, are rejected in favor of primitives that make appropriate use of pseudorandomness.

Determinism also simplifies testing. NaCl includes a battery of automated tests shared with eBACS (ECRYPT Benchmarking of Cryptographic Systems), an online cryptographic speed-measurement site [11] designed by the first two authors; this site has received, and systematically measured, 968 implementations of various cryptographic primitives from more than 100 people. The test battery found, for example, that software for a cipher named Dragon was sometimes reading outside its authorized input arrays; the same software had passed previous cryptographic test batteries. All of the core NaCl functions have also been tested against pure Python implementations, some written ourselves and some contributed by Matthew Dempsky.

## 4 Speed

Cryptographic performance problems have frequently caused users to reduce their cryptographic security levels or to turn off cryptography entirely. Consider the role of performance in the following examples:

- `https://sourceforge.net/account` is protected by SSL, but `https://sourceforge.net/develop` redirects the user's web browser to `http://sourceforge.net/develop`, actively *turning off SSL* and exposing the web pages to silent modification by sniffing attackers. Cryptography that is not actually used can be viewed as the ultimate disaster, providing no more security than any of the other cryptographic disasters discussed in this paper.
- OpenSSL's AES implementations continue to use table lookups on most CPUs, rather than obviously safe bitsliced computations that would be slower on those CPUs. The table lookups have been augmented with several

complicated countermeasures that are hoped to protect against the cache-timing attacks discussed in Section 3.

- Google has begun to allow SSL for more and more services, but only with a 1024-bit RSA key, despite
  - recommendations from the RSA company to move up to at least 2048-bit RSA by the end of 2010;
  - the same recommendations from the U.S. government; and
  - analyses from 2003 concluding that 1024-bit RSA was already breakable in under a year using hardware that governments and large companies could already afford.

  See, e.g., [29] for an analysis by Shamir (the S in RSA) and Tromer; [23] for an end-of-2010 recommendation from the RSA company; and [4] for an end-of-2010 recommendation from the U.S. government.
- DNSSEC recommends, and uses, 1024-bit RSA for practically all signatures rather than 2048-bit RSA, DSA, etc.: "In terms of performance, both RSA and DSA have comparable signature generation speeds, but DSA is much slower for signature verification. Hence, RSA is the recommended algorithm. ... The choice of key size is a tradeoff between the risk of key compromise and performance. ... RSA-SHA1 (RSA-SHA-256) until 2015, 1024 bits." See [15].
- The Tor anonymity network [35] also uses 1024-bit RSA.

**Speed of NaCl.** We do not provide any low-security options in NaCl. For example, we do not allow encryption without authentication; we do not allow any data flow from secrets to load addresses or branch conditions; and we do not allow cryptographic primitives breakable in substantially fewer than $2^{128}$ operations, such as RSA-2048.

The remaining risk is that users find NaCl too slow and turn it off, replacing it with low-security cryptographic software or no cryptography at all. NaCl avoids this type of disaster by providing exceptionally high speeds. NaCl is generally much faster than previous cryptographic libraries, even if those libraries are asked for lower security levels. More to the point, NaCl can easily keep up with terrifyingly large network loads.

For example, using a single AMD Phenom II X6 1100T CPU, NaCl performs

- more than 80000 `crypto_box` operations (public-key authenticated encryption) per second;
- more than 80000 `crypto_box_open` operations (public-key authenticator verification and decryption) per second;
- more than 70000 `crypto_sign_open` operations (signature verification) per second; and
- more than 180000 `crypto_sign` operations (signature generation) per second

for any common packet size. To put these numbers in perspective, imagine a connection flooded with 50-byte packets, each requiring a `crypto_box_open`; 80000 such packets per second would consume 32 megabits per second even

without packet overhead. A lower volume of network traffic means that the CPU needs only a fraction of its time to handle the cryptography.

NaCl provides even better speeds than this, for four reasons:

- NaCl uses a single public-key operation for a packet of any size, allowing large packets to be handled with very fast secret-key cryptography; 80000 1500-byte packets per second would fill up a gigabit-per-second link.
- A single public-key operation is shared by many packets from the same public key, allowing all the packets to be handled with very fast secret-key cryptography, if the caller splits `crypto_box` into `crypto_box_beforenm` and `crypto_box_afternm`.
- NaCl uses "encrypt-then-MAC", so forged packets are rejected without being decrypted; a flood of forgeries thus has even more trouble consuming CPU time.
- The signature system in NaCl supports fast batch verification, effectively doubling the speed of verifying a stream of valid signatures.

Most of these speedups do not reduce the cost of handling forgeries under *new* public keys, but a flooded server can continue providing very fast service to public keys that are *already known*.

The optimized implementations in the current version of NaCl are aimed at large CPUs, but all of the cryptographic primitives in NaCl can fit onto much smaller CPUs: there are no requirements for large tables or complicated code. NaCl also makes quite efficient use of bandwidth: as mentioned earlier, public keys are only 32 bytes, signed messages are only 64 bytes longer than unsigned messages, and authenticated ciphertexts are only 16 bytes longer than plaintexts.

## 5 Cryptographic primitives in NaCl

Stevens, Sotirov, Appelbaum, Lenstra, Molnar, Osvik, and de Weger announced in 2008 (see [33] and [34]) that, by exploiting various weaknesses that had been discovered in the MD5 hash function, they had created a rogue CA certificate. They could, if they wanted, have impersonated any SSL site on the Internet.

This type of disaster, cryptographic primitives being broken, is sometimes claimed to be prevented by cryptographic standardization. However, there are many examples of standards that have been publicly broken, including DES, 512-bit RSA, and these MD5-based certificates. More to the point, there are some existing standards that can reach NaCl's speeds, but those standards fall far short of NaCl's security requirements.

Our main strategy for avoiding dangerous primitives in NaCl has been to *pay attention to cryptanalysis*. There is an extensive cryptanalytic literature exploring the limits of attacks on various types of cryptographic primitives; some cryptographic structures are comfortably beyond these limits, while others inspire far less confidence. This type of security evaluation is only loosely related to standardization, as illustrated by the following example: Dobbertin, Bosselaers, and Preneel wrote "It is anticipated that these techniques can be used to

produce collisions for MD5 and perhaps also for RIPEMD" in 1996 [18], eight years before collisions in MD5 (and RIPEMD) were published and a decade before most MD5-based standards were withdrawn. They recommended switching to RIPEMD-160, which fifteen years later has still not been publicly broken.

This strategy, choosing cryptographic algorithms in light of the cryptanalytic literature, has given us higher confidence in NaCl's cryptographic primitives than in most standards. At the same time this strategy has given us the flexibility needed to push NaCl to extremely high speeds, avoiding the types of disasters discussed in Section 4.

The rest of this section discusses the cryptographic primitives used in NaCl, and explains why we expect these choices to reduce the risk of cryptographic disasters. Specifically, NaCl uses elliptic-curve cryptography, not RSA; it uses an elliptic curve, Curve25519, that has several advanced security features; it uses Salsa20, not AES (although it does include an AES implementation on the side); it uses Poly1305, not HMAC; and it uses EdDSA, not ECDSA.

We are aware that many existing protocols require AES and RSA, and that taking advantage of NaCl as described in this paper requires those protocols to be upgraded. We have prioritized security over compatibility, and as a consequence have also prioritized speed over compatibility. There are other projects that have explored the extent to which speed and security can be improved without sacrificing compatibility, but NaCl is aiming at a different point in the design space, and at applications that are not well served by the existing protocols.

**Cryptographic choices in NaCl.** RSA is somewhat older than elliptic-curve cryptography: RSA was introduced in 1977, while elliptic-curve cryptography was introduced in 1985. However, RSA has shown many more weaknesses than elliptic-curve cryptography. RSA's effective security level was dramatically reduced by the linear sieve in the late 1970s, by the quadratic sieve and ECM in the 1980s, and by the number-field sieve in the 1990s. For comparison, a few attacks have been developed against some rare elliptic curves having special algebraic structures, and the amount of computer power available to attackers has predictably increased, but typical elliptic curves require just as much computer power to break today as they required twenty years ago.

IEEE P1363 standardized elliptic-curve cryptography in the late 1990s, including a stringent list of security criteria for elliptic curves. NIST used the IEEE P1363 criteria to select fifteen specific elliptic curves at five different security levels. In 2005, NSA issued a new "Suite B" standard, recommending the NIST elliptic curves (at two specific security levels) for all public-key cryptography and withdrawing previous recommendations of RSA.

Curve25519, the particular elliptic curve used in NaCl, was introduced in [6] in 2006. It follows all of the standard IEEE P1363 security criteria; it *also* satisfies new recommendations for "twist security" and "Montgomery representation" and "Edwards representation". What this means is that secure implementations of Curve25519 are considerably simpler and faster than secure implementations of (e.g.) NIST P-256; there are fewer opportunities for implementors to make

mistakes that compromise security, and mistakes are more easily caught by reviewers.

Montgomery representation allows fast single-scalar multiplication using a Montgomery ladder [26]; this is the bottleneck in Diffie–Hellman key exchange inside `crypto_box`. It was proven in [6] that this scalar-multiplication strategy removes all need to check for special cases inside elliptic-curve additions. NaCl uses a ladder of fixed length to eliminate higher-level branches. Edwards representation allows fast multi-scalar multiplication and general addition with the same advantage of not having to check for special cases. The fixed-base-point scalar multiplication involved in `crypto_sign` uses Edwards representation for additions, and eliminates higher-level branches by using a fixed sequence of 63 point additions as described in [10, Section 4].

Salsa20 [7] is a 20-round 256-bit cipher that was submitted to eSTREAM, the ECRYPT Stream Cipher Project [19], in 2005. The same project collected dozens of submissions from 97 cryptographers in 19 countries, and then hundreds of papers analyzing the submissions. Four refereed papers from 14 cryptographers studied Salsa20, culminating in a $2^{151}$-operation "attack" against 7 rounds and a $2^{249}$-operation "attack" against 8 rounds. After 3 years of review the eSTREAM committee selected a portfolio of 4 software ciphers, including Salsa20; they recommended 12 rounds of Salsa20 as having a "comfortable margin for security".

For comparison, AES is a 14-round 256-bit cipher that was standardized ten years ago. Cryptanalysis at the time culminated in a $2^{140}$-operation "attack" against 7 rounds and a $2^{204}$-operation "attack" against 8 rounds. New research this year has reported a $2^{254}$-operation "attack" against all 14 rounds, marginally exploiting the slow key expansion of AES, an issue that was avoided in newer designs such as Salsa20. (Salsa20 also has no penalty for switching keys.) Overall each round of Salsa20 appears to have similar security to each round of AES, and 20 rounds of Salsa20 provide a very solid security margin, despite being faster than 14 rounds of AES on most CPUs.

A further difficulty with AES is that it relies on lookup tables for high-speed implementations; avoiding lookup tables compromises the speed of AES on most CPUs. Recall that, as discussed in Section 3, NaCl prohibits loading data from secret addresses. We do not mean to say that AES cannot be implemented securely: the NaCl implementation of AES is the bitsliced assembly-language implementation described in [24], together with a portable C implementation following the same approach. However, we are concerned about the extent to which security for AES requires compromising speed. Salsa20 avoids these issues: it avoids all use of lookup tables.

Poly1305 is an information-theoretically secure message-authentication code introduced in [5]. Using Poly1305 with Salsa20 is guaranteed to be as secure as using Salsa20 alone, with a security gap of at most $2^{-106}$ per byte: an attacker who can break the Poly1305 authentication can also break Salsa20. HMAC does not offer a comparable guarantee.

EdDSA was introduced quite recently in [10]. It is much newer than other primitives in NaCl but is within a well-known cloud of signature systems that includes ElGamal, Schnorr, ECDSA, etc.; it combines the safest choices available within that cloud. EdDSA is like Schnorr and unlike ECDSA in that it diversifies the hash input, adding resilience against hash collisions, and in that it avoids inversions, simplifying and accelerating implementations. EdDSA differs from Schnorr in using a double-size hash function, further reducing the risk of any hash-function problems; in requiring Edwards curves, again simplifying and accelerating implementations; and in including the public key as a further input to the hash function, alleviating concerns regarding attacks targeting many keys at once. EdDSA also avoids a minor compression mechanism, as discussed in [10]; the compression mechanism is public, so it cannot improve security, and skipping it is essential for EdDSA's fast batch verification. Finally, EdDSA generates per-message secret nonces by hashing each message together with a long-term secret, rather than requiring new randomness for each message.

NaCl's implementation of `crypto_sign` *does* use lookup tables but nevertheless avoids secret indices: each lookup from the table loads all table entries and uses arithmetic to obtain the right value. For details see [10, Section 4]. NaCl's signature verification uses signed-sliding-window scalar multiplication, which takes different amounts of time depending on the scalar, but this does not create security problems and does not violate NaCl's prohibition on secret branches: the scalar is not secret.

To summarize, all of these cryptographic choices are quite conservative. We do not expect any of them to be broken until someone succeeds in building a large quantum computer; before that happens we will extend NaCl to support post-quantum cryptography.

## References

1. Martin R. Albrecht, Kenneth G. Paterson, and Gaven J. Watson. Plaintext recovery attacks against SSH. In David Evans and Andrew Myers, editors, *2009 IEEE Symposium on Security and Privacy, Proceedings*, pages 16–26. IEEE Computer Society, 2009. http://www.isg.rhul.ac.uk/~kp/SandPfinal.pdf. 3
2. J. Bacelar Almeida, Manuel Barbosa, Jorge S. Pinto, and Bárbara Vieira. Formal verification of side channel countermeasures using self-composition. *Science of Computer Programming*. http://dx.doi.org/10.1016/j.scico.2011.10.008; to appear. 3
3. Apple. iPhone end user licence agreement. Copy distributed inside each iPhone 4; transcribed at http://rxt3ch.wordpress.com/2011/09/27/iphone-end-user-liscence-agreement-quick-refrence/. 1
4. Elaine Barker, William Barker, William Burr, William Polk, and Miles Smid. Recommendation for key management—part 1: General (revised). NIST Special Publication 800-57, 2007. http://csrc.nist.gov/groups/ST/toolkit/documents/SP800-57Part1_3-8-07.pdf. 4
5. Daniel J. Bernstein. The Poly1305-AES message-authentication code. In Henri Gilbert and Helena Handschuh, editors, *Fast Software Encryption*, volume 3557 of *LNCS*, pages 32–49. Springer, 2005. http://cr.yp.to/papers.html#poly1305. 5

6. Daniel J. Bernstein. Curve25519: new Diffie-Hellman speed records. In Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin, editors, *Public Key Cryptography—PKC 2006*, volume 3958 of *LNCS*, pages 207–228. Springer, 2006. http://cr.yp.to/papers.html#curve25519. 5
7. Daniel J. Bernstein. The Salsa20 family of stream ciphers. In Matthew Robshaw and Olivier Billet, editors, *New stream cipher designs: the eSTREAM finalists*, volume 4986 of *LNCS*, pages 84–97. Springer, 2008. http://cr.yp.to/papers.html#salsafamily. 5
8. Daniel J. Bernstein. DNSCurve: Usable security for DNS, 2009. http://dnscurve.org/. 2
9. Daniel J. Bernstein. CurveCP: Usable security for the Internet, 2011. http://curvecp.org/. 2
10. Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. High-speed high-security signatures. In Bart Preneel and Tsuyoshi Takagi, editors, *CHES 2011*, volume 6917 of *LNCS*, pages 124–142. Springer, 2011. 1, 5
11. Daniel J. Bernstein and Tanja Lange (editors). eBACS: ECRYPT benchmarking of cryptographic systems. http://bench.cr.yp.to. 3
12. Daniel Bleichenbacher. Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS#1. In Hugo Krawczyk, editor, *Advances in Cryptology—CRYPTO '98*, volume 1462 of *LNCS*, pages 1–12. Springer, 1998. http://www.bell-labs.com/user/bleichen/papers/pkcs.ps. 3
13. Billy Bob Brumley and Nicola Tuveri. Remote timing attacks are still practical. In Vijay Atluri and Claudia Diaz, editors, *Computer Security—ESORICS 2011*, volume 6879 of *LNCS*, pages 355–371. Springer, 2011. http://eprint.iacr.org/2011/232/. 3
14. "Bushing", Hector Martin "marcan" Cantero, Segher Boessenkool, and Sven Peter. PS3 epic fail, 2010. http://events.ccc.de/congress/2010/Fahrplan/attachments/1780_27c3_console_hacking_2010.pdf. 3
15. Ramaswamy Chandramouli and Scott Rose. Secure domain name system (DNS) deployment guide. NIST Special Publication 800-81r1, 2010. http://csrc.nist.gov/publications/nistpubs/800-81r1/sp-800-81r1.pdf. 4
16. Joan Daemen and Vincent Rijmen. AES proposal: Rijndael, version 2, 1999. http://csrc.nist.gov/archive/aes/rijndael/Rijndael-ammended.pdf. 3
17. Matthew Dempsky. OpenDNS adopts DNSCurve. http://blog.opendns.com/2010/02/23/opendns-dnscurve/. 2
18. Hans Dobbertin, Antoon Bosselaers, and Bart Preneel. RIPEMD-160: A strengthened version of RIPEMD. 1039:71–82, 1996. 5
19. ECRYPT. The eSTREAM project, 2008. http://www.ecrypt.eu.org/stream/. 5
20. Peter Gutmann. cryptlib security toolkit. http://www.cs.auckland.ac.nz/~pgut001/cryptlib/. 2
21. Peter Gutmann. cryptlib security toolkit: version 3.4.1: user's guide and manual. ftp://ftp.franken.de/pub/crypt/cryptlib/manual.pdf. 2, 3
22. Simon Josefsson. Don't return different errors depending on content of decrypted PKCS#1. Commit to the GnuTLS library, 2006. http://git.savannah.gnu.org/gitweb/?p=gnutls.git;a=commit;h=fc43c0d05ac450513b6dcb91949ab03eba49626a. 3
23. Burt Kaliski. TWIRL and RSA key size. http://web.archive.org/web/20030618141458/http://rsasecurity.com/rsalabs/technotes/twirl.html. 4

24. Emilia Käsper and Peter Schwabe. Faster and timing-attack resistant AES-GCM. In Christophe Clavier and Kris Gaj, editors, *Cryptographic Hardware and Embedded Systems—CHES 2009*, volume 5747 of *LNCS*, pages 1–17. Springer, 2009. http://cryptojedi.org/papers/#aesbs. 5
25. Adam Langley. ctgrind—checking that functions are constant time with Valgrind, 2010. https://github.com/agl/ctgrind. 3
26. Peter L. Montgomery. Speeding the Pollard and elliptic curve methods of factorization. *Mathematics of Computation*, 48(177):243–264, 1987. http://www.ams.org/journals/mcom/1987-48-177/S0025-5718-1987-0866113-7/S0025-5718-1987-0866113-7.pdf. 5
27. OpenSSL. OpenSSL: The open source toolkit for SSL/TLS. http://www.openssl.org/. 1
28. Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of AES. In David Pointcheval, editor, *Topics in Cryptology—CT-RSA 2006*, volume 3860 of *LNCS*, pages 1–20. Springer, 2006. 3
29. Adi Shamir and Eran Tromer. Factoring large numbers with the TWIRL device. In Dan Boneh, editor, *Advances in Cryptology—CRYPTO 2003*, volume 2729 of *LNCS*, pages 1–26. Springer, 2003. http://tau.ac.il/~tromer/papers/twirl.pdf. 4
30. Ivo Smits. Quicktun. http://wiki.ucis.nl/QuickTun. 2
31. Software in the Public Interest, Inc. Debian security advisory, DSA-1571-1 openssl—predictable random number generator, 2008. http://www.debian.org/security/2008/dsa-1571. 3
32. Jon A. Solworth. Ethos: an operating system which creates a culture of security. http://rites.uic.edu/~solworth/ethos.html. 2
33. Alexander Sotirov, Marc Stevens, Jacob Appelbaum, Arjen Lenstra, David Molnar, Dag Arne Osvik, and Benne de Weger. MD5 considered harmful today, 2008. http://www.win.tue.nl/hashclash/rogue-ca/. 5
34. Marc Stevens, Alexander Sotirov, Jacob Appelbaum, Arjen Lenstra, David Molnar, Dag Arne Osvik, and Benne de Weger. Short chosen-prefix collision for MD5 and the creation of a rogue CA certificate. In Shai Halevi, editor, *Advances in Cryptology—CRYPTO 2009*, volume 5677 of *LNCS*, pages 55–69. Springer, 2009. http://eprint.iacr.org/2009/111/. 5
35. Tor project: Anonymity online. https://www.torproject.org/. 2, 4
36. Eran Tromer, Dag Arne Osvik, and Adi Shamir. Efficient cache attacks on AES, and countermeasures. *Journal of Cryptology*, 23(1):37–71, 2010. 3