NaCl: a new crypto library

Tanja Lange, T. U. Eindhoven

Joint work with:
Peter Schwabe, R. U. Nijmegen

AES-128, RSA-2048, etc. are widely accepted standards.

Obviously infeasible to break by best attacks in literature.

Implementations are available in public cryptographic libraries such as OpenSSL.

Common security practice is to use those implementations.
NaCl: a new crypto library
D. J. Bernstein, U. Illinois Chicago
& T. U. Eindhoven
Tanja Lange, T. U. Eindhoven
Joint work with:
Peter Schwabe, R. U. Nijmegen

AES-128, RSA-2048, etc.
are widely accepted standards.
Obviously infeasible to break
by best attacks in literature.
Implementations are available
in public cryptographic libraries
such as OpenSSL.
Common security practice is
to use those implementations.
But cryptography is still
a disaster! Complete failures
of confidentiality and integrity.
NaCl: a new crypto library

D. J. Bernstein, U. Illinois Chicago
& T. U. Eindhoven
Tanja Lange, T. U. Eindhoven
Joint work with:
Peter Schwabe, R. U. Nijmegen

AES-128, RSA-2048, etc.
are widely accepted standards.
Obviously infeasible to break by best attacks in literature.
Implementations are available in public cryptographic libraries such as OpenSSL.

Common security practice is to use those implementations.
But cryptography is still a disaster! Complete failures of confidentiality and integrity.

We have designed+implemented a new cryptographic library, NaCl ("salt"), to address the underlying problems.
nacl.cr.yp.to: source and extensive documentation.

Acknowledgments:
code contributions from Matthew Dempsky (Mochi Media), Niels Duif (Eindhoven), Emilia Käsper (Leuven), Adam Langley (Google), Bo-Yin Yang (Academia Sinica).
AES-128, RSA-2048, etc. are widely accepted standards. Obviously infeasible to break by best attacks in literature. Implementations are available in public cryptographic libraries such as OpenSSL.

Common security practice is to use those implementations. But cryptography is still a disaster! Complete failures of confidentiality and integrity.

We have designed and implemented a new cryptographic library, NaCl (“salt”), to address the underlying problems.

nacl.cr.yp.to: source and extensive documentation.

Acknowledgments: code contributions from Matthew Dempsky (Mochi Media), Niels Duif (Eindhoven), Emilia Käsper (Leuven), Adam Langley (Google), Bo-Yin Yang (Academia Sinica).
NaCl: a new crypto library

D. J. Bernstein, U. Illinois Chicago
& T. U. Eindhoven

Tanja Lange, T. U. Eindhoven

Joint work with:
Peter Schwabe, R. U. Nijmegen

AES-128, RSA-2048, etc. are widely accepted standards.
Obviously infeasible to break by best attacks in literature.
Implementations are available in public cryptographic libraries such as OpenSSL.
Common security practice is to use those implementations.
But cryptography is still a disaster! Complete failures of confidentiality and integrity.

We have designed and implemented a new cryptographic library, NaCl (“salt”), to address the underlying problems.

nacl.cr.yp.to: source and extensive documentation.

Acknowledgments:
code contributions from Matthew Dempsky (Mochi Media), Niels Duif (Eindhoven), Emilia Käsper (Leuven), Adam Langley (Google), Bo-Yin Yang (Academia Sinica).
AES-128, RSA-2048, etc. are widely accepted standards. Obviously infeasible to break by best attacks in literature. Implementations are available in public cryptographic libraries such as OpenSSL. Common security practice is to use those implementations. But cryptography is still a disaster! Complete failures of confidentiality and integrity.

We have designed and implemented a new cryptographic library, NaCl (“salt”), to address the underlying problems. 

nacl.cr.yp.to: source and extensive documentation.

Acknowledgments: code contributions from Matthew Dempsky (Mochi Media), Niels Duif (Eindhoven), Emilia Käsper (Leuven), Adam Langley (Google), Bo-Yin Yang (Academia Sinica).
AES-128, RSA-2048, etc. are widely accepted standards. Obviously infeasible to break by best attacks in literature. Implementations are available in public cryptographic libraries such as OpenSSL. Common security practice is to use those implementations. But cryptography is still a disaster! Complete failures of confidentiality and integrity.

We have designed and implemented a new cryptographic library, NaCl ("salt"), to address the underlying problems. 

nacl.cr.yp.to: source and extensive documentation.

Acknowledgments: code contributions from Matthew Dempsky (Mochi Media), Niels Duif (Eindhoven), Emilia Käsper (Leuven), Adam Langley (Google), Bo-Yin Yang (Academia Sinica).

Most of the Internet is cryptographically unprotected. Primary goal of NaCl: Fix this. Main task: public-key authenticated encryption. Alice has a message $m$ for Bob. Uses Bob's public key and Alice's secret key to compute authenticated ciphertext $c$. Sends $c$ to Bob. Bob uses Alice's public key and Bob's secret key to verify and recover $m$.
AES-128, RSA-2048, etc.
are widely accepted standards.
Obviously infeasible to break
by best attacks in literature.
Implementations are available
in public cryptographic libraries
such as OpenSSL.
Common security practice is
to use those implementations.
But cryptography is still
disaster! Complete failures
of confidentiality and integrity.

We have designed+implemented
a new cryptographic library,
NaCl ("salt"), to address
the underlying problems.

nacl.cr.yp.to: source
and extensive documentation.

Acknowledgments:
code contributions from
Matthew Dempsky (Mochi Media),
Niels Duif (Eindhoven),
Emilia Käsper (Leuven),
Adam Langley (Google),
Bo-Yin Yang (Academia Sinica).

Most of the Internet
is cryptographically unprotected.
Primary goal of NaCl:
Main task: public
authenticated encryption.

Alice has a message m
for Bob.
Uses Bob's public key
Alice's secret key
authenticated ciphertext c.
Sends c to Bob.
Bob uses Alice's public
and Bob's secret key
verify and recover m.

We have designed and implemented a new cryptographic library, NaCl ("salt"), to address the underlying problems.

nacl.cr.yp.to: source and extensive documentation.

Acknowledgments:
code contributions from Matthew Dempsky (Mochi Media), Niels Duif (Eindhoven), Emilia Käsper (Leuven), Adam Langley (Google), Bo-Yin Yang (Academia Sinica).

Most of the Internet is cryptographically unprotected. Primary goal of NaCl: Fix this.

Main task: **public-key authenticated encryption**.

Alice has a message $m$ for Bob. Uses Bob’s public key and Alice’s secret key to compute authenticated ciphertext $c$. Sends $c$ to Bob. Bob uses Alice’s public key and Bob’s secret key to verify and recover $m$. 
We have designed and implemented a new cryptographic library, NaCl ("salt"), to address the underlying problems.

[nacl.cr.yp.to](http://nacl.cr.yp.to): source and extensive documentation.

Acknowledgments:
- code contributions from Matthew Dempsky (Mochi Media), Niels Duif (Eindhoven), Emilia Käsper (Leuven), Adam Langley (Google), Bo-Yin Yang (Academia Sinica).

Most of the Internet is cryptographically unprotected. Primary goal of NaCl: Fix this.

Main task: **public-key authenticated encryption**.

Alice has a message $m$ for Bob.

Uses Bob’s public key and Alice’s secret key to compute authenticated ciphertext $c$.

Sends $c$ to Bob.

Bob uses Alice’s public key and Bob’s secret key to verify and recover $m$. 
We have designed and implemented a new cryptographic library, NaCl (“salt”), to address underlying problems.

[nacl.cr.yp.to]: source and extensive documentation.

Acknowledgments:
Contributions from
Matthew Dempsky (Mochi Media),
Niels Duif (Eindhoven),
Emilia Käser (Leuven),
Adam Langley (Google),
Bo-Yin Yang (Academia Sinica).

Most of the Internet is cryptographically unprotected.
Primary goal of NaCl: Fix this.

Main task: **public-key authenticated encryption**.

Alice has a message \( m \) for Bob.

Uses Bob’s public key and Alice’s secret key to compute authenticated ciphertext \( c \).
Sends \( c \) to Bob.

Bob uses Alice’s public key and Bob’s secret key to verify and recover \( m \).

Alice using a typical cryptographic library:

Generate random AES key.
Use AES key to encrypt packet.
Hash encrypted packet.
Read RSA key from wire format.
Use key to sign hash.
Read Bob’s key from wire format.
Use key to encrypt signature etc.
Convert to wire format.

Plus more code:
allocate storage,
handle errors, etc.
We have designed and implemented a new cryptographic library, NaCl (“salt”), to address the underlying problems.

nacl.cr.yp.to: source and extensive documentation.

Acknowledgments: code contributions from Matthew Dempsky (Mochi Media), Niels Duif (Eindhoven), Emilia Kasper (Leuven), Adam Langley (Google), Bo-Yin Yang (Academia Sinica).

Most of the Internet is cryptographically unprotected. Primary goal of NaCl: Fix this.

Main task: **public-key authenticated encryption**.

Alice has a message $m$ for Bob.

Uses Bob’s public key and Alice’s secret key to compute authenticated ciphertext $c$.

Sends $c$ to Bob.

Bob uses Alice’s public key and Bob’s secret key to verify and recover $m$.

Alice using a typical cryptographic library:

Generate random AES key.

Use AES key to encrypt packet.

Hash encrypted packet.

Read RSA key from wire format.

Use key to sign hash.

Read Bob’s key from wire format.

Use key to encrypt signature etc.

Convert to wire format.

Plus more code: allocate storage, handle errors, etc.
We have designed and implemented a new cryptographic library, NaCl ("salt"), to address the underlying problems.

nacl.cr.yp.to: source and extensive documentation.

Acknowledgments: code contributions from Matthew Dempsky (Mochi Media), Niels Duif (Eindhoven), Emilia Käser (Leuven), Adam Langley (Google), Bo-Yin Yang (Academia Sinica).

Most of the Internet is cryptographically unprotected. Primary goal of NaCl: Fix this.

Main task: **public-key authenticated encryption**.

Alice has a message $m$ for Bob.

Uses Bob’s public key and Alice’s secret key to compute authenticated ciphertext $c$.

Sends $c$ to Bob.

Bob uses Alice’s public key and Bob’s secret key to verify and recover $m$.

Alice using a typical cryptographic library:

Generate random AES key.
Use AES key to encrypt packet.
Hash encrypted packet.
Read RSA key from wire format.
Use key to sign hash.
Read Bob’s key from wire format.
Use key to encrypt signature etc.
Convert to wire format.

Plus more code: allocate storage, handle errors, etc.
Most of the Internet is cryptographically unprotected. Primary goal of NaCl: Fix this.

Main task: **public-key authenticated encryption**.

Alice has a message \( m \) for Bob.

Uses Bob’s public key and Alice’s secret key to compute authenticated ciphertext \( c \).

Sends \( c \) to Bob.

Bob uses Alice’s public key and Bob’s secret key to verify and recover \( m \).

Alice using a typical cryptographic library:

Generate random AES key.
Use AES key to encrypt packet.
Hash encrypted packet.
Read RSA key from wire format.
Use key to sign hash.
Read Bob’s key from wire format.
Use key to encrypt signature etc.
Convert to wire format.

Plus more code:
allocate storage, handle errors, etc.
Most of the Internet is cryptographically unprotected. Primary goal of NaCl: Fix this.

Main task: public-key authenticated encryption.

Alice has a message $m$ for Bob. Uses Bob’s public key and Alice’s secret key to compute authenticated ciphertext $c$. Sends $c$ to Bob.

Alice using a typical cryptographic library:
Generate random AES key.
Use AES key to encrypt packet.
Hash encrypted packet.
Read RSA key from wire format.
Use key to sign hash.
Read Bob’s key from wire format.
Use key to encrypt signature etc.
Convert to wire format.

Plus more code: allocate storage, handle errors, etc.

Alice using NaCl:

c = crypto_box(m,n,pk,sk)
Most of the Internet is cryptographically unprotected. NaCl: Fix this. Main task: public-key authenticated encryption.

Alice has a message $m$ for Bob. Uses Bob’s public key and Alice’s secret key to compute authenticated ciphertext $c$. Sends $c$ to Bob. Bob uses Alice’s public key and Bob’s secret key to verify and recover $m$.

Alice using a typical cryptographic library:
- Generate random AES key.
- Use AES key to encrypt packet.
- Hash encrypted packet.
- Read RSA key from wire format.
- Use key to sign hash.
- Read Bob’s key from wire format.
- Use key to encrypt signature etc.
- Convert to wire format.

Plus more code:
- allocate storage,
- handle errors, etc.

Alice using NaCl:
- $c = \text{crypto\_box}(m, \text{pk}, \text{sk})$
Most of the Internet is cryptographically unprotected. Primary goal of NaCl: Fix this.

Main task: public-key authenticated encryption.

Alice has a message $m$ for Bob. Uses Bob's public key and Alice's secret key to compute authenticated ciphertext $c$. Sends $c$ to Bob. Bob uses Alice's public key and Bob's secret key to verify and recover $m$.

Alice using a typical cryptographic library:
- Generate random AES key.
- Use AES key to encrypt packet.
- Hash encrypted packet.
- Read RSA key from wire format.
- Use key to sign hash.
- Read Bob's key from wire format.
- Use key to encrypt signature etc.
- Convert to wire format.

Plus more code:
- allocate storage,
- handle errors, etc.

Alice using NaCl:
```c
 Alice using NaCl:
c = crypto_box(m, n, pk, sk)
```
Alice using a typical cryptographic library:

Generate random AES key.
Use AES key to encrypt packet.
Hash encrypted packet.
Read RSA key from wire format.
Use key to sign hash.
Read Bob’s key from wire format.
Use key to encrypt signature etc.
Convert to wire format.

Plus more code:
allocate storage,
handle errors, etc.

Alice using NaCl:
\[ c = \text{crypto}_\text{box}(m,n,pk,sk) \]
Alice using a typical cryptographic library:

Generate random AES key.
Use AES key to encrypt packet.
Hash encrypted packet.
Read RSA key from wire format.
Use key to sign hash.
Read Bob’s key from wire format.
Use key to encrypt signature etc.
Convert to wire format.

Plus more code:
allocate storage,
handle errors, etc.

Alice using NaCl:

\[ c = \text{crypto\_box}(m, n, pk, sk) \]

32-byte secret key \( sk \).
32-byte public key \( pk \).
24-byte nonce \( n \).
c is 16 bytes longer than \( m \).

All objects are C++ std::string variables represented in wire format, ready for storage/transmission.

C NaCl: similar, using pointers; no memory allocation, no failures.
Alice using a cryptographic library:
- Generate random AES key.
- Use AES key to encrypt packet.
- Hash encrypted packet.
- Read RSA key from wire format.
- Use key to sign hash.
- Read Bob’s key from wire format.
- Use key to encrypt signature etc.
- Convert to wire format.
- More code:
  - Allocate storage,
  - Handle errors, etc.

Alice using NaCl:
```
c = crypto_box(m, n, pk, sk)
```
- 32-byte secret key sk.
- 32-byte public key pk.
- 24-byte nonce n.
- c is 16 bytes longer than m.

All objects are C++ `std::string` variables represented in wire format, ready for storage/transmission.

C NaCl: similar, using pointers; no memory allocation, no failures.

Bob verifying, decrypting:
```
m = crypto_box_open(c, n, pk, sk)
```
Initial key generation:
```
pk = crypto_box_keypair(&sk)
```
Alice using NaCl:
\[ c = \text{crypto\_box}(m,n,pk,sk) \]

32-byte secret key \( sk \).
32-byte public key \( pk \).
24-byte nonce \( n \).
\( c \) is 16 bytes longer than \( m \).

All objects are C++
std::string variables
represented in wire format,
ready for storage/transmission.

C NaCl: similar, using pointers;
no memory allocation, no failures.

Bob verifying, decrypting:
\[ m = \text{crypto\_box\_open}(c,n,pk,sk) \]

Initial key generation:
\[ pk = \text{crypto\_box\_keypair}(&sk) \]
Alice using NaCl:
\[ c = \text{crypto\_box}(m, n, pk, sk) \]
32-byte secret key \( sk \).
32-byte public key \( pk \).
24-byte nonce \( n \).
\( c \) is 16 bytes longer than \( m \).

All objects are C++
\text{std::string} variables
represented in wire format,
ready for storage/transmission.

C NaCl: similar, using pointers;
no memory allocation, no failures.

Bob verifying, decrypting:
\[ m = \text{crypto\_box\_open}(c, n, pk, sk) \]
Initial key generation:
\[ pk = \text{crypto\_box\_keypair}() \]
Alice using NaCl:
c = crypto_box(m,n,pk,sk)
32-byte secret key sk.
32-byte public key pk.
24-byte nonce n.
c is 16 bytes longer than m.

All objects are C++
std::string variables
represented in wire format,
ready for storage/transmission.

C NaCl: similar, using pointers;
no memory allocation, no failures.

Bob verifying, decrypting:
m=crypto_box_open(c,n,pk,sk)
Initial key generation:
pk = crypto_box_keypair(&sk)
Alice using NaCl:
c = crypto_box(m,n,pk,sk)
32-byte secret key sk.
32-byte public key pk.
24-byte nonce n.
c is 16 bytes longer than m.

All objects are C++
std::string variables
represented in wire format,
ready for storage/transmission.

C NaCl: similar, using pointers;
no memory allocation, no failures.

Bob verifying, decrypting:
m=crypto_box_open(c,n,pk,sk)
Initial key generation:
pk = crypto_box_keypair(&sk)
Can instead use signatures
for public messages:
pk = crypto_sign_keypair(&sk)
64-byte secret key,
32-byte public key.

sm = crypto_sign(m,sk)
64 bytes overhead.
m = crypto_sign_open(sm,pk)
Alice using NaCl:
c = crypto_box(m,n,pk,sk)

- 32-byte secret key \( sk \).
- 32-byte public key \( pk \).
- 24-byte nonce \( n \).
- \( c \) is 16 bytes longer than \( m \).

All objects are C++ std::string variables represented in wire format, ready for storage/transmission.

C NaCl: similar, using pointers; no memory allocation, no failures.

Bob verifying, decrypting:
m=crypto_box_open(c,n,pk,sk)

Initial key generation:
pk = crypto_box_keypair(&sk)

Can instead use signatures for public messages:
pk = crypto_sign_keypair(&sk)

- 64-byte secret key,
- 32-byte public key.

sm = crypto_sign(m,sk)

- 64 bytes overhead.
m = crypto_sign_open(sm,pk)

“This sounds too simple! Don’t applications need more?”
Alice using NaCl:
\[ c = \text{crypto\_box}(m, n, pk, sk) \]

32-byte secret key \( sk \).
32-byte public key \( pk \).
24-byte nonce \( n \).

\( c \) is 16 bytes longer than \( m \).

All objects are C++ `std::string` variables represented in wire format, ready for storage/transmission.

C NaCl: similar, using pointers; no memory allocation, no failures.

Bob verifying, decrypting:
\[ m = \text{crypto\_box\_open}(c, n, pk, sk) \]

Initial key generation:
\[ pk = \text{crypto\_box\_keypair}(&sk) \]

Can instead use **signatures** for public messages:
\[ pk = \text{crypto\_sign\_keypair}(&sk) \]

64-byte secret key, 32-byte public key.

\[ sm = \text{crypto\_sign}(m, sk) \]
64 bytes overhead.

\[ m = \text{crypto\_sign\_open}(sm, pk) \]

“This sounds too simple! Don’t applications need more?”
Alice using NaCl: 
\[ c = \text{crypto\_box}(m,n,pk,sk) \]

32-byte secret key \( sk \).
32-byte public key \( pk \).
24-byte nonce \( n \).
\( c \) is 16 bytes longer than \( m \).
All objects are C++ std::string variables represented in wire format, ready for storage/transmission.

C NaCl: similar, using pointers; no memory allocation, no failures.

Bob verifying, decrypting:
\[ m = \text{crypto\_box\_open}(c,n,pk,sk) \]

Initial key generation:
\[ pk = \text{crypto\_box\_keypair}(&sk) \]

Can instead use **signatures** for public messages:
\[ pk = \text{crypto\_sign\_keypair}(&sk) \]
64-byte secret key,
32-byte public key.
\[ sm = \text{crypto\_sign}(m,sk) \]
64 bytes overhead.
\[ m = \text{crypto\_sign\_open}(sm,pk) \]

“This sounds too simple! Don’t applications need more?”
Bob verifying, decrypting:
\[ m = \text{crypto\_box\_open}(c,n,pk,sk) \]

Initial key generation:
\[ pk = \text{crypto\_box\_keypair}(\&sk) \]

Can instead use **signatures** for public messages:
\[ pk = \text{crypto\_sign\_keypair}(\&sk) \]

64-byte secret key, 32-byte public key.

\[ sm = \text{crypto\_sign}(m,sk) \]

64 bytes overhead.

\[ m = \text{crypto\_sign\_open}(sm,pk) \]

“This sounds too simple! Don’t applications need more?”
Bob verifying, decrypting:
m = crypto_box_open(c, n, pk, sk)

Initial key generation:
pk = crypto_box_keypair(&sk)

Can instead use **signatures** for public messages:
pk = crypto_sign_keypair(&sk)

64-byte secret key,
32-byte public key.

sm = crypto_sign(m, sk)
64 bytes overhead.

m = crypto_sign_open(sm, pk)

“This sounds too simple!
Don’t applications need more?”

Examples of applications using NaCl’s crypto_box:
DNSCurve and DNSCrypt, high-security authenticated encryption for DNS queries; deployed by OpenDNS.
QUIC, Google’s TLS replacement.
MinimaLT in Ethos OS, faster TLS replacement.
Threema, encrypted-chat app.
Bob verifying, decrypting:
\texttt{crypto_box_open(c,n,pk,sk)}

Key generation:
\texttt{crypto_box_keypair(&sk)}

Instead use \textbf{signatures}
for public messages:
\texttt{crypto_sign_keypair(&sk)}

64-byte secret key,
32-byte public key.

\texttt{crypto_sign(m,sk)}
64 bytes overhead.

\texttt{crypto_sign_open(sm,pk)}

“This sounds too simple! Don’t applications need more?”

Examples of applications using NaCl’s \texttt{crypto_box}:
DNSCurve and DNSCrypt, high-security authenticated encryption for DNS queries; deployed by OpenDNS.

QUIC, Google’s TLS replacement.

MinimaLT in Ethos OS, faster TLS replacement.

Threema, encrypted-chat app.

Related projects:
Various ports, repackaging, language bindings, etc.: e.g.,
\url{github.com/jedisct1/libsodium}

TweetNaCl: NaCl in 100 tweets; on the path towards full audit.
Bernstein, van Gastel, Janssen, Lange, Smetsers.

\url{tweetnacl.cr.yp.to}
\url{twitter.com/tweetnacl}

Benchmarking of > 1000 crypto implementations using same API:
\url{bench.cr.yp.to}
Verifying, decrypting:

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

Initial key generation:

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

Can instead use signatures for public messages:

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

64-byte secret key, 32-byte public key.

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

64 bytes overhead.

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

“This sounds too simple! Don’t applications need more?”

Examples of applications using NaCl’s `crypto_box`:

- DNSCurve and DNSCrypt, high-security authenticated encryption for DNS queries; deployed by OpenDNS.
- QUIC, Google’s TLS replacement.
- MinimaLT in Ethos OS, faster TLS replacement.
- Threema, encrypted-chat app.

Related projects

Various ports, repackaging, language bindings, etc.: e.g., 
github.com/jedisct1/libsodium

TweetNaCl: NaCl in 100 tweets; on the path towards full audit.

Bernstein, van Gastel, Janssen, Lange, Schwabe, Smetsers.

tweetnacl.cr.yp.to
twitter.com/tweetnacl

Benchmarking of > 1000 crypto implementations using same API:
bench.cr.yp.to
Bob verifying, decrypting:
\[ m = \text{crypto\_box\_open}(c, n, \text{pk}, \text{sk}) \]

Initial key generation:
\[ \text{pk} = \text{crypto\_box\_keypair}(\&\text{sk}) \]
Can instead use signatures for public messages:
\[ \text{pk} = \text{crypto\_sign\_keypair}(\&\text{sk}) \]
64-byte secret key, 32-byte public key.
\[ \text{sm} = \text{crypto\_sign}(m, \text{sk}) \]
64 bytes overhead.
\[ m = \text{crypto\_sign\_open}(\text{sm}, \text{pk}) \]

“This sounds too simple! Don’t applications need more?”

Examples of applications using NaCl’s `crypto_box`:
- DNSCurve and DNSCrypt, high-security authenticated encryption for DNS queries; deployed by OpenDNS.
- QUIC, Google’s TLS replacement.
- MinimaLT in Ethos OS, faster TLS replacement.
- Threema, encrypted-chat app.

Related projects
Various ports, repackaging, language bindings, etc.: e.g.,
github.com/jedisct1/libsodium
TweetNaCl: NaCl in 100 tweets; on the path towards full audit.
Bernstein, van Gastel, Janssen, Lange, Schwabe, Smetsers.
tweetnacl.cr.yp.to
twitter.com/tweetnacl

Benchmarking of >1000 crypto implementations using same API:
bench.cr.yp.to
“This sounds too simple! Don’t applications need more?”

Examples of applications using NaCl’s crypto_box:

- DNSCurve and DNSCrypt, high-security authenticated encryption for DNS queries; deployed by OpenDNS.
- QUIC, Google’s TLS replacement.
- MinimaLT in Ethos OS, faster TLS replacement.
- Threema, encrypted-chat app.

Related projects

Various ports, repackaging, language bindings, etc.: e.g.,

- [github.com/jedisct1/libsodium](https://github.com/jedisct1/libsodium)
- TweetNaCl: NaCl in 100 tweets; on the path towards full audit. Bernstein, van Gastel, Janssen, Lange, Schwabe, Smetsers.
- [tweetnacl.cr.yp.to](https://tweetnacl.cr.yp.to)
- [twitter.com/tweetnacl](https://twitter.com/tweetnacl)

Benchmarkeding of >1000 crypto implementations using same API:

- [bench.cr.yp.to](https://bench.cr.yp.to)
This sounds too simple! Applications need more?" Examples of applications using NaCl's crypto_box: DnSCurve and DNSCrypt, high-security authenticated encryption for DNS queries; deployed by OpenDNS. QUIC, Google's TLS replacement. MinimaLT in Ethos OS, faster TLS replacement. Threema, encrypted-chat app.

Related projects
Various ports, repackaging, language bindings, etc.: e.g., github.com/jedisct1/libsodium
TweetNaCl: NaCl in 100 tweets; on the path towards full audit. Bernstein, van Gastel, Janssen, Lange, Schwabe, Smetsers. tweetnacl.cr.yp.to twitter.com/tweetnacl

Benchmarking of >1000 crypto implementations using same API: bench.cr.yp.to

No secret load addresses
2005 Osvik–Shamir–Tromer: 65ms to steal Linux AES key used for hard-disk encryption. Attack process on same CPU but without privileges. Almost all AES implementations use fast lookup tables. Kernel’s secret AES key influences table-load addresses, influencing CPU cache state, influencing measurable timings of the attack. 65ms to compute influence.
This sounds too simple! Don't applications need more?

Examples of applications using NaCl's `crypto_box`:
- DNSCurve and DNSCrypt, high-security authenticated encryption for DNS queries; deployed by OpenDNS.
- QUIC, Google's TLS replacement.
- MinimaLT in Ethos OS, faster TLS replacement.
- Threema, encrypted-chat app.

Related projects
Various ports, repackaging, language bindings, etc.: e.g.,
github.com/jedisct1/libsodium
TweetNaCl: NaCl in 100 tweets; on the path towards full audit.
Bernstein, van Gastel, Janssen, Lange, Schwabe, Smetsers.
tweetnacl.cr.yp.to
twitter.com/tweetnacl

Benchmarks of >1000 crypto implementations using same API:
bench.cr.yp.to

No secret load addresses
Attack process on same CPU but without privileges.
Almost all AES implementations use fast lookup tables. Kernel's secret AES key influences table-lookup; influencing CPU cache state; influencing measurable timings of the attack process.

65ms to compute influence
− 1
.
Related projects

Various ports, repackaging, language bindings, etc.: e.g., github.com/jedisct1/libsodium

TweetNaCl: NaCl in 100 tweets; on the path towards full audit.
Bernstein, van Gastel, Janssen, Lange, Schwabe, Smetsers.
tweetnacl.cr.yp.to
twitter.com/tweetnacl

Benchmarking of >1000 crypto implementations using same API: bench.cr.yp.to


No secret load addresses

2005 Osvik–Shamir–Tromer:
65ms to steal Linux AES key used for hard-disk encryption.
Attack process on same CPU but without privileges.

Almost all AES implementations use fast lookup tables.
Kernel’s secret AES key influences table-load addresses, influencing CPU cache state, influencing measurable timings of the attack process.
65ms to compute influence—
Related projects

Various ports, repackaging, language bindings, etc.: e.g.,
github.com/jedisct1/libsodium

TweetNaCl: NaCl in 100 tweets; on the path towards full audit.
Bernstein, van Gastel, Janssen, Lange, Schwabe, Smetsers.
tweetnacl.cr.yp.to
twitter.com/tweetnacl

Benchmarking of >1000 crypto implementations using same API:
bench.cr.yp.to

No secret load addresses


Almost all AES implementations use fast lookup tables. Kernel’s secret AES key influences table-load addresses, influencing CPU cache state, influencing measurable timings of the attack process.

65ms to compute influence $^{-1}$. 
Related projects
Various ports, repackaging, language bindings, etc.: e.g.,
github.com/jedisct1/libsodium
TweetNaCl: NaCl in 100 tweets; on the path towards full audit.
Bernstein, van Gastel, Janssen, Lange, Schwabe, Smetsers.
tweetnacl.cr.yp.to
twitter.com/tweetnacl

Benchmarking of >1000 crypto implementations using same API:
bench.cr.yp.to

No secret load addresses


Almost all AES implementations use fast lookup tables. Kernel’s secret AES key influences table-load addresses, influencing CPU cache state, influencing measurable timings of the attack process. 65ms to compute influence$^{-1}$.

Most cryptographic libraries still use secret load addresses but add “countermeasures” intended to obscure influence upon the CPU cache state. Not confidence-inspiring; likely to be breakable.
No secret load addresses


Almost all AES implementations use fast lookup tables.

Kernel’s secret AES key influences table-load addresses, influencing CPU cache state, influencing measurable timings of the attack process.

65ms to compute influence.$^{-1}$.

Most cryptographic libraries still use secret load addresses but add “countermeasures” intended to obscure influence upon the CPU cache state. Not confidence-inspiring; likely to be breakable.
No secret load addresses

Attack process on same CPU but without privileges.

Almost all AES implementations use fast lookup tables.
Kernel’s secret AES key influences table-load addresses, influencing CPU cache state, influencing measurable timings of the attack process. 65ms to compute influence$^{-1}$. 

Most cryptographic libraries still use secret load addresses but add “countermeasures” intended to obscure influence upon the CPU cache state. Not confidence-inspiring; likely to be breakable.
No secret load addresses


Almost all AES implementations use fast lookup tables. Kernel’s secret AES key influences table-load addresses, influencing CPU cache state, influencing measurable timings of the attack process. 65ms to compute influence\(^{-1}\).

Most cryptographic libraries still use secret load addresses but add “countermeasures” intended to obscure influence upon the CPU cache state. Not confidence-inspiring; likely to be breakable.
No secret load addresses


Almost all AES implementations use fast lookup tables. Kernel’s secret AES key influences table-load addresses, influencing CPU cache state, influencing measurable timings of the attack process. 65ms to compute influence$^{-1}$.

Most cryptographic libraries still use secret load addresses but add “countermeasures” intended to obscure influence upon the CPU cache state. Not confidence-inspiring; likely to be breakable.

NaCl systematically avoids all loads from addresses that depend on secret data. Eliminates this type of disaster.

Timing attack+defense tutorial: Schwabe talk tomorrow 11:00.
No secret load addresses

2005 Osvik–Shamir–Tromer: 65ms to steal Linux AES key used for hard-disk encryption. Attack process on same CPU but without privileges. Almost all AES implementations use fast lookup tables. Kernel's secret AES key influences table-load addresses, influencing CPU cache state, influencing measurable timings of the attack process. 65ms to compute influence\(^{-1}\).

Most cryptographic libraries still use secret load addresses but add “countermeasures” intended to obscure influence upon the CPU cache state. Not confidence-inspiring; likely to be breakable.

NaCl systematically avoids all loads from addresses that depend on secret data. Eliminates this type of disaster.

Timing attack+defense tutorial: Schwabe talk tomorrow 11:00.

No secret branch conditions

2011 Brumley–Tuveri: minutes to steal another machine's OpenSSL ECDSA key. Secret branch conditions influence timings. Most cryptographic software has many small-scale variations: e.g., memcmp for IPsec MACs.
Most cryptographic libraries still use secret load addresses but add “countermeasures” intended to obscure influence upon the CPU cache state. Not confidence-inspiring; likely to be breakable.

NaCl systematically avoids all loads from addresses that depend on secret data. Eliminates this type of disaster.

Timing attack+defense tutorial: Schwabe talk tomorrow 11:00.
Most cryptographic libraries still use secret load addresses but add “countermeasures” intended to obscure influence upon the CPU cache state. Not confidence-inspiring; likely to be breakable.

NaCl systematically avoids all loads from addresses that depend on secret data. Eliminates this type of disaster.

Timing attack+defense tutorial: Schwabe talk tomorrow 11:00.
Most cryptographic libraries still use secret load addresses but add “countermeasures” intended to obscure influence upon the CPU cache state. Not confidence-inspiring; likely to be breakable.

NaCl systematically avoids all loads from addresses that depend on secret data. Eliminates this type of disaster.

Timing attack+defense tutorial: Schwabe talk tomorrow 11:00.

No secret branch conditions

2011 Brumley–Tuveri: minutes to steal another machine’s OpenSSL ECDSA key. Secret branch conditions influence timings.

Most cryptographic software has many more small-scale variations in timing: e.g., memcmp for IPsec MACs.
Most cryptographic libraries still use secret load addresses but add “countermeasures” intended to obscure influence upon the CPU cache state. Not confidence-inspiring; likely to be breakable.

NaCl systematically avoids all loads from addresses that depend on secret data. Eliminates this type of disaster.

Timing attack+defense tutorial: Schwabe talk tomorrow 11:00.

No secret branch conditions

2011 Brumley–Tuveri: minutes to steal another machine’s OpenSSL ECDSA key. Secret branch conditions influence timings.

Most cryptographic software has many more small-scale variations in timing: e.g., memcmp for IPsec MACs.

NaCl systematically avoids all branch conditions that depend on secret data. Eliminates this type of disaster.
Most cryptographic libraries still use secret load addresses but add “countermeasures” intended to obscure influence upon the CPU cache state. Confidence-inspiring; likely to be breakable.

NaCl systematically avoids all loads from addresses that depend on secret data. Eliminates this type of disaster.

Timing attack+defense tutorial: Schwabe talk tomorrow 11:00.

No secret branch conditions

2011 Brumley–Tuveri: minutes to steal another machine’s OpenSSL ECDSA key. Secret branch conditions influence timings.

Most cryptographic software has many more small-scale variations in timing: e.g., memcmp for IPsec MACs.

NaCl systematically avoids all branch conditions that depend on secret data. Eliminates this type of disaster.

No padding oracles

1998 Bleichenbacher: Decrypt SSL RSA ciphertext by observing server responses to $\approx 10^6$ variants of ciphertext.

SSL first inverts RSA, then checks for “PKCS padding” (which many forgeries have). Subsequent processing applies more serious integrity checks.

Server responses reveal pattern of PKCS forgeries; pattern reveals plaintext.
Most cryptographic libraries still use secret load addresses but add “countermeasures” intended to obscure influence upon the CPU cache state. Not confidence-inspiring; likely to be breakable.

NaCl systematically avoids all loads from addresses that depend on secret data. Eliminates this type of disaster.

Timing attack+defense tutorial: Schwabe talk tomorrow 11:00.

No secret branch conditions

2011 Brumley–Tuveri: minutes to steal another machine’s OpenSSL ECDSA key. Secret branch conditions influence timings.

Most cryptographic software has many more small-scale variations in timing: e.g., memcmp for IPsec MACs.

NaCl systematically avoids all branch conditions that depend on secret data. Eliminates this type of disaster.

No padding oracles

1998 Bleichenbacher: Decrypt SSL RSA ciphertext by observing server responses to $\approx 10^6$ variants of ciphertext. SSL first inverts RSA, then checks for “PKCS padding” (which many forgeries have). Subsequent processing applies more serious integrity checks. Server responses reveal pattern of PKCS forgeries; pattern reveals plaintext.
No secret branch conditions

2011 Brumley–Tuveri: minutes to steal another machine’s OpenSSL ECDSA key. Secret branch conditions influence timings.

Most cryptographic software has many more small-scale variations in timing:
e.g., memcmp for IPsec MACs.

NaCl systematically avoids all branch conditions that depend on secret data. Eliminates this type of disaster.

No padding oracles

1998 Bleichenbacher: Decrypt SSL RSA ciphertext by observing server responses to ≈$10^6$ variants of ciphertext. SSL first inverts RSA, then checks for “PKCS padding” (which many forgeries have). Subsequent processing applies more serious integrity checks.

Server responses reveal pattern of PKCS forgeries; pattern reveals plaintext.
No secret branch conditions

2011 Brumley–Tuveri: minutes to steal another machine’s OpenSSL ECDSA key. Secret branch conditions influence timings.

Most cryptographic software has many more small-scale variations in timing: e.g., memcmp for IPsec MACs.

NaCl systematically avoids all branch conditions that depend on secret data. Eliminates this type of disaster.

No padding oracles

1998 Bleichenbacher: Decrypt SSL RSA ciphertext by observing server responses to ≈10^6 variants of ciphertext. SSL first inverts RSA, then checks for “PKCS padding” (which many forgeries have). Subsequent processing applies more serious integrity checks.

Server responses reveal pattern of PKCS forgeries; pattern reveals plaintext.
No secret branch conditions

2011 Brumley–Tuveri: minutes to steal another machine’s OpenSSL ECDSA key. Secret branch conditions influence timings.

Most cryptographic software has many more small-scale variations in timing: \texttt{memcmp} for IPsec MACs.

NaCl systematically avoids all branch conditions that depend on secret data. Eliminates this type of disaster.

No padding oracles

1998 Bleichenbacher: Decrypt SSL RSA ciphertext by observing server responses to $\approx 10^6$ variants of ciphertext.

SSL first inverts RSA, then checks for “PKCS padding” (which many forgeries have). Subsequent processing applies more serious integrity checks.

Server responses reveal pattern of PKCS forgeries; pattern reveals plaintext.

Typical defense strategy: try to hide differences between padding checks and subsequent integrity checks.

But hard to get this right: see, e.g., Lucky 13 and POODLE.
No secret branch conditions


Most cryptographic software has many more small-scale variations in timing: e.g., `memcmp` for IPsec MACs.

NaCl systematically avoids all branch conditions that depend on secret data. Eliminates this type of disaster.

---

No padding oracles

1998 Bleichenbacher: Decrypt SSL RSA ciphertext by observing server responses to $\approx10^6$ variants of ciphertext.

SSL first inverts RSA, then checks for “PKCS padding” (which many forgeries have). Subsequent processing applies more serious integrity checks.

Server responses reveal pattern of PKCS forgeries; pattern reveals plaintext.

Typical defense strategy: try to hide differences between padding checks and subsequent integrity checks.

But hard to get this right: see, e.g., Lucky 13 and POODLE.
No secret branch conditions
2011 Brumley–Tuveri: minutes to steal another machine's OpenSSL ECDSA key. Secret branch conditions influence timings. Most cryptographic software has many more small-scale variations in timing: e.g., memcmp for IPsec MACs. NaCl systematically avoids all branch conditions that depend on secret data. Eliminates this type of disaster.

No padding oracles
1998 Bleichenbacher: Decrypt SSL RSA ciphertext by observing server responses to $\approx 10^6$ variants of ciphertext. SSL first inverts RSA, then checks for “PKCS padding” (which many forgeries have). Subsequent processing applies more serious integrity checks. Server responses reveal pattern of PKCS forgeries; pattern reveals plaintext.

Typical defense strategy: try to hide differences between padding checks and subsequent integrity checks. But hard to get this right: see, e.g., Lucky 13 and POODLE.
No padding oracles

1998 Bleichenbacher:
Decrypt SSL RSA ciphertext by observing server responses to \( \approx 10^6 \) variants of ciphertext.

SSL first inverts RSA, then checks for “PKCS padding” (which many forgeries have). Subsequent processing applies more serious integrity checks.

Server responses reveal pattern of PKCS forgeries; pattern reveals plaintext.

Typical defense strategy: try to hide differences between padding checks and subsequent integrity checks.

But hard to get this right: see, e.g., Lucky 13 and POODLE.
No padding oracles

1998 Bleichenbacher:
Decrypt SSL RSA ciphertext
by observing server responses
to $\approx 10^6$ variants of ciphertext.

SSL first inverts RSA,
then checks for “PKCS padding”
(which many forgeries have).
Subsequent processing applies
more serious integrity checks.

Server responses reveal
pattern of PKCS forgeries;
pattern reveals plaintext.

Typical defense strategy:
try to hide differences
between padding checks and
subsequent integrity checks.

But hard to get this right:
see, e.g., Lucky 13 and POODLE.

NaCl does not decrypt
unless message is authenticated.
Verification procedure rejects
all forgeries in constant time.
Attacks are further constrained
by per-nonce key separation
and standard nonce handling.
Bleichenbacher: Decrypt SSL RSA ciphertext by observing server responses to $10^6$ variants of ciphertext.

SSL first inverts RSA, then checks for “PKCS padding” (which many forgeries have). Subsequent processing applies more serious integrity checks. Server responses reveal pattern of PKCS forgeries; pattern reveals plaintext.

Typical defense strategy: try to hide differences between padding checks and subsequent integrity checks. But hard to get this right: see, e.g., Lucky 13 and POODLE.

NaCl does not decrypt unless message is authenticated. Verification procedure rejects all forgeries in constant time. Attacks are further constrained by per-nonce key separation and standard nonce handling.

Centralizing randomness: Debian/Ubuntu OpenSSL keys for 1.5 years had only 15 bits of entropy. Debian developer had removed a subtle line of OpenSSL randomness-generating code.
1998 Bleichenbacher: Decrypt SSL RSA ciphertext by observing server responses to $10^6$ variants of ciphertext. SSL first inverts RSA, then checks for "PKCS padding" (which many forgeries have). Subsequent processing applies more serious integrity checks. Server responses reveal pattern of PKCS forgeries; pattern reveals plaintext.

Typical defense strategy: try to hide differences between padding checks and subsequent integrity checks.

But hard to get this right: see, e.g., Lucky 13 and POODLE. NaCl does not decrypt unless message is authenticated. Verification procedure rejects all forgeries in constant time. Attacks are further constrained by per-nonce key separation and standard nonce handling.

2008 Bello: Debian/Ubuntu OpenSSL keys for 1.5 years had only 15 bits of entropy. Debian developer had removed a subtle line of OpenSSL randomness-generating code.

Centralizing randomness
Typical defense strategy: try to hide differences between padding checks and subsequent integrity checks.

But hard to get this right: see, e.g., Lucky 13 and POODLE.

NaCl does not decrypt unless message is authenticated. Verification procedure rejects all forgeries in constant time. Attacks are further constrained by per-nonce key separation and standard nonce handling.

Centralizing randomness
2008 Bello: Debian/Ubuntu OpenSSL keys for 1.5 years had only 15 bits of entropy.

Debian developer had removed a subtle line of OpenSSL randomness-generating code.
Typical defense strategy: try to hide differences between padding checks and subsequent integrity checks.

But hard to get this right: see, e.g., Lucky 13 and POODLE.

NaCl does not decrypt unless message is authenticated. Verification procedure rejects all forgeries in constant time. Attacks are further constrained by per-nonce key separation and standard nonce handling.

Centralizing randomness
2008 Bello: Debian/Ubuntu OpenSSL keys for 1.5 years had only 15 bits of entropy.

Debian developer had removed a subtle line of OpenSSL randomness-generating code.
Typical defense strategy: try to hide differences between padding checks and subsequent integrity checks.

But hard to get this right: see, e.g., Lucky 13 and POODLE.

NaCl does not decrypt unless message is authenticated. Verification procedure rejects all forgeries in constant time. Attacks are further constrained by per-nonce key separation and standard nonce handling.

Centralizing randomness

2008 Bello: Debian/Ubuntu OpenSSL keys for 1.5 years had only 15 bits of entropy.

Debian developer had removed a subtle line of OpenSSL randomness-generating code.

NaCl uses /dev/urandom, the OS random-number generator. Reviewing this kernel code is much more tractable than reviewing separate RNG code in every security library.
Typical defense strategy: try to hide differences between padding checks and subsequent integrity checks. But hard to get this right: see, e.g., Lucky 13 and POODLE.

NaCl does not decrypt unless message is authenticated. Verification procedure rejects forgeries in constant time. Attacks are further constrained by per-nonce key separation and standard nonce handling.

Centralizing randomness

2008 Bello: Debian/Ubuntu OpenSSL keys for 1.5 years had only 15 bits of entropy.

Debian developer had removed a subtle line of OpenSSL randomness-generating code.

NaCl uses /dev/urandom, the OS random-number generator. Reviewing this kernel code is much more tractable than reviewing separate RNG code in every security library.

Centralizing randomness

2008 Bello: Debian/Ubuntu OpenSSL keys for 1.5 years had only 15 bits of entropy.

Debian developer had removed a subtle line of OpenSSL randomness-generating code.

NaCl uses /dev/urandom, the OS random-number generator. Reviewing this kernel code is much more tractable than reviewing separate RNG code in every security library.

Centralizing randomness

2008 Bello: Debian/Ubuntu OpenSSL keys for 1.5 years had only 15 bits of entropy.

Debian developer had removed a subtle line of OpenSSL randomness-generating code.

NaCl uses /dev/urandom, the OS random-number generator. Reviewing this kernel code is much more tractable than reviewing separate RNG code in every security library.
Centralizing randomness

2008 Bello: Debian/Ubuntu OpenSSL keys for 1.5 years had only 15 bits of entropy.

Debian developer had removed a subtle line of OpenSSL randomness-generating code.

NaCl uses /dev/urandom, the OS random-number generator.

NaCl does not decrypt unless message is authenticated.

Verification procedure rejects all forgeries in constant time.

Attacks are further constrained by per-nonce key separation and standard nonce handling.

Centralization allows OS to merge many entropy sources into a pool feeding many applications.

Merging is deterministic and auditable. Can survive many bad/failing/malicious sources if there is one good one.
Typical defense strategy:
try to hide differences between padding checks and subsequent integrity checks.
But hard to get this right:
see, e.g., Lucky 13 and POODLE.
NaCl does not decrypt unless message is authenticated.
Verification procedure rejects all forgeries in constant time.
Attacks are further constrained by per-nonce key separation and standard nonce handling.

Centralizing randomness
2008 Bello: Debian/Ubuntu OpenSSL keys for 1.5 years had only 15 bits of entropy.
Debian developer had removed a subtle line of OpenSSL randomness-generating code.
NaCl uses /dev/urandom, the OS random-number generator.
Reviewing this kernel code is much more tractable than reviewing separate RNG code in every security library.

Centralization allows OS to merge many entropy sources into pool feeding many applications.
Merging is deterministic and auditable. Can survive many bad/failing/malicious sources if there is one good source.
Centralizing randomness

2008 Bello: Debian/Ubuntu OpenSSL keys for 1.5 years had only 15 bits of entropy.

Debian developer had removed a subtle line of OpenSSL randomness-generating code.

NaCl uses /dev/urandom, the OS random-number generator. Reviewing this kernel code is much more tractable than reviewing separate RNG code in every security library.

Centralization allows OS to merge many entropy sources into pool feeding many applications.

Merging is deterministic and auditable. Can survive many bad/failing/malicious sources if there is one good source.
Centralizing randomness

2008 Bello: Debian/Ubuntu OpenSSL keys for 1.5 years had only 15 bits of entropy.

Debian developer had removed a subtle line of OpenSSL randomness-generating code.

NaCl uses /dev/urandom, the OS random-number generator. Reviewing this kernel code is much more tractable than reviewing separate RNG code in every security library.

Centralization allows OS to merge many entropy sources into pool feeding many applications.

Merging is deterministic and auditable. Can survive many bad/failing/malicious sources if there is one good source.

Huge step backwards: Intel’s RDRAND in applications. Single entropy source; no backup; likely to be poorly cloned; backdoorable (CHES 2013); non-auditable. Not used in NaCl.
Centralizing randomness

2008 Bello: Debian/Ubuntu OpenSSL keys for 1.5 years had only 15 bits of entropy. Debian developer had removed a subtle line of OpenSSL randomness-generating code. NaCl uses /dev/urandom, the OS random-number generator. Reviewing this kernel code is much more tractable than reviewing separate RNG code in every security library.

Centralization allows OS to merge many entropy sources into pool feeding many applications. Merging is deterministic and auditable. Can survive many bad/failing/malicious sources if there is one good source. Huge step backwards: Intel’s RDRAND in applications. Single entropy source; no backup; likely to be poorly cloned; backdoorable (CHES 2013); non-auditable. Not used in NaCl.

Avoiding unnecessary randomness

<table>
<thead>
<tr>
<th>Buolmness</th>
<th>Avoiding unnecessary randomness</th>
</tr>
</thead>
</table>

Centralization allows OS to merge many entropy sources into pool feeding many applications.

Merging is deterministic and auditable. Can survive many bad/failing/malicious sources if there is one good source.

Huge step backwards:
- Intel’s RDRAND in applications.
- Single entropy source; no backup;
- likely to be poorly cloned;
- backdoorable (CHES 2013);
- non-auditable. Not used in NaCl.

Avoiding unnecessary randomness

Centralization allows OS to merge many entropy sources into pool feeding many applications.

Merging is deterministic and auditable. Can survive many bad/failing/malicious sources if there is one good source.

Huge step backwards:
- Intel’s RDRAND in applications.
- Single entropy source; no backup;
- likely to be poorly cloned;
- backdoorable (CHES 2013);
- non-auditable. Not used in NaCl.
Centralizing randomness

2008 Bello: Debian/Ubuntu OpenSSL keys for 1.5 years had only 15 bits of entropy. Debian developer had removed a subtle line of OpenSSL randomness-generating code.

NaCl uses /dev/urandom, the OS random-number generator. Reviewing this kernel code is much more tractable than reviewing separate RNG code in every security library.

Centralization allows OS to merge many entropy sources into pool feeding many applications. Merging is deterministic and auditable. Can survive many bad/failing/malicious sources if there is one good source.

Huge step backwards: Intel's RDRAND in applications. Single entropy source; no backup; likely to be poorly cloned; backdoorable (CHES 2013); non-auditable. Not used in NaCl.

Avoiding unnecessary randomness

Centralization allows OS to merge many entropy sources into pool feeding many applications. Merging is deterministic and auditable. Can survive many bad/failing/malicious sources if there is one good source.

Huge step backwards:
Intel’s RDRAND in applications.
Single entropy source; no backup;
likely to be poorly cloned;
backdoorable (CHES 2013);
non-auditable. Not used in NaCl.

Avoiding unnecessary randomness
Centralization allows OS to merge many entropy sources into pool feeding many applications. Merging is deterministic and auditable. Can survive many bad/failing/malicious sources if there is one good source.

Huge step backwards: Intel’s RDRAND in applications. Single entropy source; no backup; likely to be poorly cloned; backdoorable (CHES 2013); non-auditable. Not used in NaCl.

Avoiding unnecessary randomness

NaCl has deterministic crypto_box and crypto_sign. Randomness only for keypair. Eliminates this type of disaster.

Also simplifies testing. NaCl uses automated test battery from bench.cr.yp.to.
Centralization allows OS to merge many entropy sources into a pool feeding many applications. Merging is deterministic and auditable. Can survive many bad/failing/malicious sources if there is one good source.

Huge step backwards: Intel's RDRAND in applications. Single entropy source; no backup; likely to be poorly cloned; backdoorable (CHES 2013); non-auditable. Not used in NaCl.

Avoiding unnecessary randomness

NaCl has deterministic crypto_box and crypto_sign. Randomness only for keypair. Eliminates this type of disaster.

Also simplifies testing. NaCl uses automated test battery from bench.cr.yp.to.

Avoiding pure crypto failures
Centralization allows the OS to merge many entropy sources into a pool feeding many applications. Merging is deterministic and auditable. It can survive many bad/failing/malicious sources if there is one good source.

Huge step backwards: Intel's RDRAND in applications. Single entropy source; no backup; likely to be poorly cloned; backdoorable (CHES 2013); non-auditable. Not used in NaCl.

Avoiding unnecessary randomness:


NaCl has deterministic crypto_box and crypto_sign. Randomness only for keypair. Eliminates this type of disaster. Also simplifies testing. NaCl uses automated test battery from bench.cr.yp.to.

Avoiding pure crypto failures:

Centralization allows OS to merge many entropy sources into a pool feeding many applications. Merging is deterministic and auditable. Can survive many bad/failing/malicious sources if there is one good source.

Huge step backwards: Intel's RDRAND in applications. Single entropy source; no backup; likely to be poorly cloned; backdoorable (CHES 2013); non-auditable. Not used in NaCl.

Avoiding unnecessary randomness

NaCl has deterministic crypto_box and crypto_sign. Randomness only for keypair. Eliminates this type of disaster.

Also simplifies testing. NaCl uses automated test battery from bench.cr.yp.to.

Avoiding pure crypto failures
Avoiding unnecessary randomness


NaCl has *deterministic* crypto_box and crypto_sign. Randomness only for keypair. Eliminates this type of disaster.

Also simplifies testing. NaCl uses automated test battery from bench.cr.yp.to.

Avoiding pure crypto failures

Avoiding unnecessary randomness

NaCl has deterministic crypto_box and crypto_sign. Randomness only for keypair. Eliminates this type of disaster.

Also simplifies testing. NaCl uses automated test battery from bench.cr.yp.to.

Avoiding pure crypto failures
2012 Flame: new MD5 attack.
Avoiding unnecessary randomness

2010 Bushing–Marcan–Segher–Sven: Sony ignored ECDSA requirement of new randomness for each signature. \( \Rightarrow \) Signatures leaked PS3 code-signing key.

NaCl has *deterministic* crypto_box and crypto_sign. Randomness only for keypair. Eliminates this type of disaster.

Also simplifies testing. NaCl uses automated test battery from bench.cr.yp.to.

Avoiding pure crypto failures


2012 Flame: new MD5 attack.

Fact: By 1996, a few years after the introduction of MD5, Preneel and Dobbertin were calling for MD5 to be scrapped.

NaCl *pays attention to cryptanalysis* and makes very conservative choices of cryptographic primitives.
Avoiding unnecessary randomness

Bushing–Marcan–Segher–Sven: Sony ignored ECDSA requirement of new randomness for each signature. ⇒ Signatures leaked PS3 code-signing key.

NaCl has deterministic crypto_box and crypto_sign. Randomness only for keypair. Eliminates this type of disaster. Also simplifies testing. NaCl uses automated test battery from bench.cr.yp.to.

Avoiding pure crypto failures


2012 Flame: new MD5 attack.

Fact: By 1996, a few years after the introduction of MD5, Preneel and Dobbertin were calling for MD5 to be scrapped.

NaCl pays attention to cryptanalysis and makes very conservative choices of cryptographic primitives.

Speed

Crypto performance problems often lead users to reduce cryptographic security levels or give up on cryptography.

Example 1: Google SSL used RSA-1024 until 2013.

Security note: Analyses in 2003 concluded that RSA-1024 was breakable; e.g., 2003 Shamir–Tromer estimated 1 year, ≈ $10^7$ USD.

RSA Labs and NIST response: Move to RSA-2048 by 2010.
Avoiding unnecessary randomness


NaCl has deterministic crypto_box and crypto_sign. Randomness only for keypair. Eliminates this type of disaster. Also simplifies testing. NaCl uses automated test battery from bench.cr.yp.to.

Avoiding pure crypto failures

2012 Flame: new MD5 attack.

Fact: By 1996, a few years after the introduction of MD5, Preneel and Dobbertin were calling for MD5 to be scrapped. NaCl pays attention to cryptanalysis and makes very conservative choices of cryptographic primitives.

Speed

Crypto performance problems often lead users to reduce cryptographic security levels or give up on cryptography.

Example 1: Google SSL used RSA-1024 until 2013.

Security note:
Analyses in 2003 concluded that RSA-1024 was breakable; e.g., 2003 Shamir–Tromer estimated 1 year, ≈ 10^7 USD.

RSA Labs and NIST response: Move to RSA-2048 by 2010.
Avoiding unnecessary randomness


NaCl has deterministic crypto_box and crypto_sign. Randomness only for keypair. Eliminates this type of disaster. Also simplifies testing. NaCl uses automated test battery from bench.cr.yp.to.

Avoiding pure crypto failures


2012 Flame: new MD5 attack.

Fact: By 1996, a few years after the introduction of MD5, Preneel and Dobbertin were calling for MD5 to be scrapped.

NaCl pays attention to cryptanalysis and makes very conservative choices of cryptographic primitives.

Speed

Crypto performance problems often lead users to reduce cryptographic security levels or give up on cryptography.

Example 1: Google SSL used RSA-1024 until 2013.

Security note:
Analyses in 2003 concluded that RSA-1024 was breakable, e.g., 2003 Shamir–Tromer estimated 1 year, ≈10^7 USD. RSA Labs and NIST response: Move to RSA-2048 by 2010.
Avoiding pure crypto failures
2012 Flame: new MD5 attack.
Fact: By 1996, a few years after the introduction of MD5, Preneel and Dobbertin were calling for MD5 to be scrapped.
NaCl pays attention to cryptanalysis and makes very conservative choices of cryptographic primitives.

Speed
Crypto performance problems often lead users to reduce cryptographic security levels or give up on cryptography.
Example 1: Google SSL used RSA-1024 until 2013.
Security note: Analyses in 2003 concluded that RSA-1024 was breakable; e.g., 2003 Shamir–Tromer estimated 1 year, $\approx 10^7$ USD.
RSA Labs and NIST response: Move to RSA-2048 by 2010.
Avoiding pure crypto failures


Fame: new MD5 attack.

By 1996, a few years after the introduction of MD5, Preneel and Dobbertin were calling for MD5 to be scrapped.

NaCl pays attention to cryptanalysis and makes very conservative choices of cryptographic primitives.

Speed

Crypto performance problems often lead users to reduce cryptographic security levels or give up on cryptography.

Example 1: Google SSL used RSA-1024 until 2013.

Security note: Analyses in 2003 concluded that RSA-1024 was breakable; e.g., 2003 Shamir–Tromer estimated 1 year, ≈ 10^7 USD.

RSA Labs and NIST response: Move to RSA-2048 by 2010.

Example 2: Tor used RSA-1024 until 2013 switch to Curve25519.

Example 3: DNSSEC uses RSA-1024: “tradeoff between the risk of key compromise and performance”.

Example 4: OpenSSL on ARM uses secret AES load addresses.

Example 5: https://sourceforge.net/account is protected by SSL but https://sourceforge.net/develop turns off crypto: redirects to http://sourceforge.net/develop.
Avoiding pure crypto failures


2012 Flame: new MD5 attack.

Fact: By 1996, a few years after the introduction of MD5, Preneel and Dobbertin were calling for MD5 to be scrapped.

NaCl pays attention to cryptanalysis and makes very conservative choices of cryptographic primitives.

---

Speed

Crypto performance problems often lead users to reduce cryptographic security levels or give up on cryptography.

Example 1: Google SSL used RSA-1024 until 2013.

Security note:
Analyses in 2003 concluded that RSA-1024 was breakable; e.g., 2003 Shamir–Tromer estimated 1 year, \( \approx 10^7 \) USD.

RSA Labs and NIST response: Move to RSA-2048 by 2010.

Example 2: Tor used RSA-1024 until 2013 switch to Curve25519.

Example 3: DNSSEC uses RSA-1024: “tradeoff between the risk of key compromise and performance...”

Example 4: OpenSSL on ARM uses secret AES load addresses.

Example 5: https://sourceforge.net/account is protected by SSL but https://sourceforge.net/develop turns off crypto: redirects to http://sourceforge.net/develop.
Avoiding pure crypto failures


2012 Flame: new MD5 attack.

Fact: By 1996, a few years after the introduction of MD5, Preneel and Dobbertin were calling for MD5 to be scrapped.

NaCl pays attention to cryptanalysis and makes very conservative choices of cryptographic primitives.

Example 1: Google SSL used RSA-1024 until 2013.

Security note:
Analyses in 2003 concluded that RSA-1024 was breakable; e.g., 2003 Shamir–Tromer estimated 1 year, $\approx 10^7$ USD.

RSA Labs and NIST response: Move to RSA-2048 by 2010.

Example 2: Tor used RSA-1024 until 2013 switch to Curve25519.

Example 3: DNSSEC uses RSA-1024: “tradeoff between the risk of key compromise and performance...”

Example 4: OpenSSL on ARM uses secret AES load addresses.

Example 5:
Speed

Crypto performance problems often lead users to reduce cryptographic security levels or give up on cryptography.

Example 1: Google SSL used RSA-1024 until 2013.

Security note:
Analyses in 2003 concluded that RSA-1024 was breakable; e.g., 2003 Shamir–Tromer estimated 1 year, $\approx 10^7$ USD. RSA Labs and NIST response: Move to RSA-2048 by 2010.

Example 2: Tor used RSA-1024 until 2013 switch to Curve25519.

Example 3: DNSSEC uses RSA-1024: “tradeoff between the risk of key compromise and performance. . . .”

Example 4: OpenSSL on ARM uses secret AES load addresses.

Example 5: 
performance problems had users to reduce cryptographic security levels or give up on cryptography.

Example 1: Google SSL used RSA-1024 until 2013. 

Note: Analyses in 2003 concluded RSA-1024 was breakable; e.g., 2003 Shamir–Tromer estimated 1 year, \(\approx 10^7\) USD.

NaCl has no low-security options. e.g. crypto_box always encrypts and authenticates. e.g. no RSA-1024; not even RSA-2048.

Example 2: Tor used RSA-1024 until 2013 switch to Curve25519.

Example 3: DNSSEC uses RSA-1024: “tradeoff between the risk of key compromise and performance. . .”

Example 4: OpenSSL on ARM uses secret AES load addresses.

Example 5: 
https://sourceforge.net/account is protected by SSL but 
Speed

Crypto performance problems often lead users to reduce cryptographic security levels or give up on cryptography.

Example 1: Google SSL used RSA-1024 until 2013.

Security note: Analyses in 2003 concluded that RSA-1024 was breakable; e.g., 2003 Shamir–Tromer estimated 1 year, \(\approx 10^7\) USD.

RSA Labs and NIST response: Move to RSA-2048 by 2010.

Example 2: Tor used RSA-1024 until 2013 switch to Curve25519.

Example 3: DNSSEC uses RSA-1024: “tradeoff between the risk of key compromise and performance...”

Example 4: OpenSSL on ARM uses secret AES load addresses.

Example 5: NaCl has no low-security options.

e.g. crypto_box always encrypts and authenticates.

e.g. no RSA-1024; not even RSA-2048.

Example 2: Tor used RSA-1024 until 2013 switch to Curve25519.

Example 3: DNSSEC uses RSA-1024: “tradeoff between the risk of key compromise and performance...”

Example 4: OpenSSL on ARM uses secret AES load addresses.

Example 5:
https://sourceforge.net/account is protected by SSL but
https://sourceforge.net/develop turns off crypto: redirects to
http://sourceforge.net/develop.

NaCl has no low-security options. e.g. crypto_box always encrypts and authenticates.
e.g. no RSA-1024; not even RSA-2048.
Example 2: Tor used RSA-1024 until 2013 switch to Curve25519.

Example 3: DNSSEC uses RSA-1024: “tradeoff between the risk of key compromise and performance…”

Example 4: OpenSSL on ARM uses secret AES load addresses.

Example 5:
https://sourceforge.net/account is protected by SSL but

NaCl has no low-security options.
e.g. crypto_box always encrypts and authenticates.
e.g. no RSA-1024; not even RSA-2048.
Example 2: Tor used RSA-1024 until 2013 switch to Curve25519.

Example 3: DNSSEC uses RSA-1024: “tradeoff between the risk of key compromise and performance…”

Example 4: OpenSSL on ARM uses secret AES load addresses.

Example 5:

NaCl has no low-security options. e.g. crypto_box always encrypts and authenticates. e.g. no RSA-1024; not even RSA-2048.

Remaining risk:
Users find NaCl too slow ⇒ switch to low-security libraries or disable crypto entirely.
Example 2: Tor used RSA-1024 until 2013 switch to Curve25519.

Example 3: DNSSEC uses RSA-1024: “tradeoff between the risk of key compromise and performance...”

Example 4: OpenSSL on ARM uses secret AES load addresses.

Example 5: 
https://sourceforge.net/account is protected by SSL but 
https://sourceforge.net/develop turns off crypto: redirects to 
http://sourceforge.net/develop.

NaCl has no low-security options. e.g. crypto_box always encrypts and authenticates. e.g. no RSA-1024; not even RSA-2048.

Remaining risk:
Users find NaCl too slow ⇒ switch to low-security libraries or disable crypto entirely.

How NaCl avoids this risk:
NaCl is exceptionally fast. Much faster than other libraries. Keeps up with the network.
Example 2: Tor used RSA-1024 until 2013 switch to Curve25519.

Example 3: DNSSEC uses RSA-1024: “tradeoff between the risk of key compromise and performance...”

Example 4: OpenSSL on ARM uses secret AES load addresses.

Example 5:

NaCl has no low-security options. e.g. crypto_box always encrypts and authenticates.

NaCl operations per second for any common packet size, using AMD Phenom II X6 1100T CPU ($190 in 2011):
crypto_box : > 80000.
crypto_box_open : > 80000.
crypto_sign_open : > 70000.
crypto_sign : > 180000.

Remaining risk:
Users find NaCl too slow ⇒ switch to low-security libraries or disable crypto entirely.

How NaCl avoids this risk:
NaCl is exceptionally fast. Much faster than other libraries. Keeps up with the network.
Example 2: Tor used RSA-1024 until 2013 switch to Curve25519.

Example 3: DNSSEC uses RSA-1024: "tradeoff between the risk of key compromise and performance..."

Example 4: OpenSSL on ARM uses secret AES load addresses.

Example 5: https://sourceforge.net/account is protected by SSL but https://sourceforge.net/develop turns off crypto: redirects to http://sourceforge.net/develop.

NaCl has no low-security options.
e.g. crypto_box always encrypts and authenticates.
e.g. no RSA-1024;
not even RSA-2048.

Remaining risk:
Users find NaCl too slow ⇒ switch to low-security libraries or disable crypto entirely.

How NaCl avoids this risk:
NaCl is exceptionally fast.
Much faster than other libraries.
*Keeps up with the network.*

NaCl operations per second for any common packet size, using AMD Phenom II X6 1100T CPU ($190 in 2011):
crypto_box: > 80000.
crypto_box_open: > 80000.
crypto_sign_open: > 70000.
crypto_sign: > 180000.
NaCl has no low-security options. e.g. crypto_box always encrypts and authenticates.
e.g. no RSA-1024; not even RSA-2048.

Remaining risk:
Users find NaCl too slow ⇒ switch to low-security libraries or disable crypto entirely.

How NaCl avoids this risk:
NaCl is exceptionally fast. Much faster than other libraries. Keeps up with the network.
NaCl has no low-security options. e.g. crypto_box always encrypts and authenticates. e.g. no RSA-1024; not even RSA-2048.

Remaining risk:
Users find NaCl too slow ⇒ switch to low-security libraries or disable crypto entirely.

How NaCl avoids this risk:
NaCl is exceptionally fast. Much faster than other libraries. *Keeps up with the network.*

NaCl operations per second for any common packet size, using AMD Phenom II X6 1100T CPU ($190 in 2011):
crypto_box: $>80000$.
crypto_box_open: $>80000$.
crypto_sign_open: $>70000$.
crypto_sign: $>180000$. 

NaCl has no low-security options. e.g. crypto_box always encrypts and authenticates. e.g. no RSA-1024; not even RSA-2048.

Remaining risk: Users find NaCl too slow ⇒ switch to low-security libraries or disable crypto entirely.

How NaCl avoids this risk: NaCl is exceptionally fast. Much faster than other libraries. *Keeps up with the network.*

NaCl operations per second for any common packet size, using AMD Phenom II X6 1100T CPU ($190 in 2011):

- crypto_box: >80000.
- crypto_box_open: >80000.
- crypto_sign_open: >70000.
- crypto_sign: >180000.

Handles arbitrary packet floods up to ≈30 Mbps per CPU, depending on protocol details.
NaCl has no low-security options. Crypto_box always encrypts and authenticates. E.g. no RSA-1024; even RSA-2048.

Remaining risk:
NaCl too slow $\Rightarrow$ switch to low-security libraries or disable crypto entirely.

NaCl avoids this risk: exceptionally fast. Much faster than other libraries. Keep up with the network.

NaCl operations per second for any common packet size, using AMD Phenom II X6 1100T CPU ($190 in 2011)$: 

```
crypto_box: >80000.
crypto_box_open: >80000.
crypto_sign_open: >70000.
crypto_sign: >180000.
```

Handles arbitrary packet floods up to $\approx$ 30 Mbps per CPU, depending on protocol details.

But wait, it's even faster!
1. Pure secret-key crypto for any packet size: 80000 1500-byte packets/second fill up a 1 Gbps link.
2. Pure secret-key crypto for many packets from same public key, if application splits crypto_box into crypto_box_beforenm and crypto_box_afternm.
NaCl has no low-security options. e.g. crypto_box always encrypts and authenticates. e.g. no RSA-1024; not even RSA-2048. Remaining risk: Users find NaCl too slow ⇒ switch to low-security libraries or disable crypto entirely. How NaCl avoids this risk: NaCl is exceptionally fast. Much faster than other libraries. Keeps up with the network.

NaCl operations per second for any common packet size, using AMD Phenom II X6 1100T CPU ($190 in 2011):
crypto_box: >80000.
crypto_box_open: >80000.
crypto_sign_open: >70000.
crypto_sign: >180000.

Handles arbitrary packet floods up to ≈30 Mbps per CPU, depending on protocol details.

But wait, it’s even faster!
1. Pure secret-key crypto for any packet size: 80000 1500-byte packets/second fill up a 1 Gbps link.
2. Pure secret-key crypto for many packets from same public key, if application splits crypto_box into crypto_box_beforenm and crypto_box_afternm.
NaCl has no low-security options. e.g. crypto_box always encrypts and authenticates. e.g. no RSA-1024; not even RSA-2048. Remaining risk: Users find NaCl too slow ⇒ switch to low-security libraries or disable crypto entirely.

How NaCl avoids this risk: NaCl is exceptionally fast. Much faster than other libraries. Keeps up with the network.

<table>
<thead>
<tr>
<th>NaCl operations per second for any common packet size, using AMD Phenom II X6 1100T CPU ($190 in 2011):</th>
</tr>
</thead>
<tbody>
<tr>
<td>crypto_box: &gt;80000.</td>
</tr>
<tr>
<td>crypto_box_open: &gt;80000.</td>
</tr>
<tr>
<td>crypto_sign_open: &gt;70000.</td>
</tr>
<tr>
<td>crypto_sign: &gt;180000.</td>
</tr>
</tbody>
</table>

Handles arbitrary packet floods up to ≈30 Mbps per CPU, depending on protocol details.

But wait, it’s even faster!

1. Pure secret-key crypto for any packet size: 80000 1500-byte packets/second fill up a 1 Gbps link.

2. Pure secret-key crypto for many packets from same public key, if application splits crypto_box into crypto_box_beforenm and crypto_box_afternm.
NaCl operations per second for any common packet size, using AMD Phenom II X6 1100T CPU ($190 in 2011):

- `crypto_box`: >80000.
- `crypto_box_open`: >80000.
- `crypto_sign_open`: >70000.
- `crypto_sign`: >180000.

Handles arbitrary packet floods up to ≈30 Mbps per CPU, depending on protocol details.

But wait, it’s even faster!

1. Pure secret-key crypto for any packet size:
   80000 1500-byte packets/second fill up a 1 Gbps link.

2. Pure secret-key crypto for many packets from same public key, if application splits `crypto_box` into `crypto_box_beforenm` and `crypto_box_afternm`. 
But wait, it’s even faster!

1. Pure secret-key crypto for any packet size: 80000 1500-byte packets/second fill up a 1 Gbps link.

2. Pure secret-key crypto for many packets from same public key, if application splits crypto_box into crypto_box_beforenm and crypto_box_afternm.

3. Very fast rejection of forged packets under known public keys: no time spent on decryption.
   (This doesn’t help much for forgeries under new keys, but flooded server can continue providing fast service to known keys.)

4. Fast batch verification, doubling speed of crypto_sign_open for valid signatures.
NaCl operations per second for any common packet size, using AMD Phenom II X6 1100T CPU ($190 in 2011):

- **crypto_box**: >80000.
- **crypto_box_open**: >80000.
- **crypto_sign_open**: >70000.
- **crypto_sign**: >180000.

Handles arbitrary packet floods up to \( \approx 30 \text{ Mbps} \) per CPU, depending on protocol details.

But wait, it’s even faster!

1. Pure secret-key crypto for any packet size:
   80000 1500-byte packets/second fill up a 1 Gbps link.

2. Pure secret-key crypto for many packets from same public key, if application splits
   crypto_box into crypto_box_beforenm and crypto_box_afternm.

3. Very fast rejection of forged packets under known public keys:
   no time spent on decryption.
   (This doesn’t help for forgeries under new keys, but flooded server can continue providing fast service to *known* keys.)

4. Fast batch verification, doubling speed of crypto_sign_open for valid signatures.
NaCl operations per second for any common packet size, using AMD Phenom II X6 1100T CPU ($190 in 2011):
crypto_box : > 80000.
crypto_box_open : > 80000.
crypto_sign_open : > 70000.
crypto_sign : > 180000.

Handles arbitrary packet floods up to ≈ 30 Mbps per CPU, depending on protocol details.

But wait, it’s even faster!

1. Pure secret-key crypto for any packet size:
   80000 1500-byte packets/second fill up a 1 Gbps link.

2. Pure secret-key crypto for many packets from same public key, if application splits
   crypto_box into
   crypto_box_beforenm and crypto_box_afternm.

3. Very fast rejection of forged packets under known public keys:
   no time spent on decryption.
   (This doesn’t help much for forgeries under new keys, but flooded server can continue providing fast service to known keys.)

4. Fast batch verification, doubling speed of
   crypto_sign_open for valid signatures.
But wait, it’s even faster!

1. Pure secret-key crypto for any packet size:
   80000 1500-byte packets/second fill up a 1 Gbps link.

2. Pure secret-key crypto for many packets from same public key, if application splits
   crypto_box into crypto_box_beforenm and crypto_box_afternm.

3. Very fast rejection of forged packets under known public keys:
   no time spent on decryption.
   (This doesn’t help much for forgeries under new keys, but flooded server can continue providing fast service to known keys.)

4. Fast batch verification, doubling speed of crypto_sign_open for valid signatures.
But wait, it’s even faster!

1. Pure secret-key crypto for any packet size: 80,000 1500-byte packets/second fill up a 1 Gbps link.

2. Pure secret-key crypto for many packets from same public key, if application splits crypto_box into crypto_box_beforenm and crypto_box_afternm.

3. Very fast rejection of forged packets under known public keys: no time spent on decryption.
   (This doesn’t help much for forgeries under new keys, but flooded server can continue providing fast service to known keys.)

4. Fast batch verification, doubling speed of crypto_sign_open for valid signatures.

Also fast on small devices.

“NEON crypto” (CHES 2012) on 1GHz ARM Cortex-A8 core: 498349 cycles (2000/second) + 7.78 cycles/byte (1 Gbps) for box; and for verify: 624846 cycles (1600/second).
But wait, it's even faster!

1. Pure secret-key crypto for any packet size:
   80000 1500-byte packets/second fill up a 1 Gbps link.

2. Pure secret-key crypto for many packets from same public key, if application splits crypto_box into crypto_box_beforenm and crypto_box_afternm.

3. Very fast rejection of forged packets under known public keys: no time spent on decryption. (This doesn’t help much for forgeries under new keys, but flooded server can continue providing fast service to known keys.)

4. Fast batch verification, doubling speed of crypto_sign_open for valid signatures.

Also fast on small devices. “NEON crypto” (CHES 2012) on 1GHz ARM Cortex-A8 core:
498349 cycles (2000/second) + 7.78 cycles/byte for box; and for verify:
624846 cycles (1600/second).
3. Very fast rejection of forged packets under known public keys: no time spent on decryption.
   (This doesn’t help much for forgeries under *new* keys, but flooded server can continue providing fast service to *known* keys.)

4. Fast batch verification, doubling speed of `crypto_sign_open` for valid signatures.
3. Very fast rejection of forged packets under known public keys: no time spent on decryption. (This doesn’t help much for forgeries under new keys, but flooded server can continue providing fast service to known keys.)

4. Fast batch verification, doubling speed of crypto_sign_open for valid signatures.

Also fast on small devices.

“NEON crypto” (CHES 2012) on 1GHz ARM Cortex-A8 core: 498349 cycles (2000/second) + 7.78 cycles/byte (1 Gbps) for box; and for verify: 624846 cycles (1600/second).
3. Very fast rejection of forged packets under known public keys: no time spent on decryption.

(This doesn’t help much for forgeries under new keys, but flooded server can continue providing fast service to known keys.)

4. Fast batch verification, doubling speed of crypto_sign_open for valid signatures. Also fast on small devices.

“NEON crypto” (CHES 2012) on 1GHz ARM Cortex-A8 core: 498,349 cycles (2000/second) + 7.78 cycles/byte (1 Gbps) for box; and for verify: 624,846 cycles (1600/second).

1GHz Cortex-A8 was high-end smartphone core in 2010: e.g., Samsung Exynos 3110 (Galaxy S); TI OMAP3630 (Motorola Droid X); Apple A4 (iPad 1/iPhone 4).
3. Very fast rejection of forged packets under known public keys: no time spent on decryption.

(This doesn’t help much for forgeries under new keys, but flooded server can continue providing fast service to known keys.)

4. Fast batch verification, doubling speed of crypto_sign_open for valid signatures.

Also fast on small devices.

“NEON crypto” (CHES 2012) on 1GHz ARM Cortex-A8 core: 498349 cycles (2000/second) + 7.78 cycles/byte (1 Gbps) for box; and for verify: 624846 cycles (1600/second).

1GHz Cortex-A8 was high-end smartphone core in 2010: e.g., Samsung Exynos 3110 (Galaxy S); TI OMAP3630 (Motorola Droid X); Apple A4 (iPad 1/iPhone 4).

2013: Allwinner A13, $5 in bulk.
3. Very fast rejection of forged packets under known public keys: no time spent on decryption. (This doesn't help much for forgeries under new keys, but flooded server can still providing fast service to known keys.)

4. Fast batch verification, doubling speed of crypto_sign_open for valid signatures.

Also fast on small devices.

“NEON crypto” (CHES 2012) on 1GHz ARM Cortex-A8 core: 498349 cycles (2000/second) + 7.78 cycles/byte (1 Gbps) for box; and for verify: 624846 cycles (1600/second).

1GHz Cortex-A8 was high-end smartphone core in 2010: e.g., Samsung Exynos 3110 (Galaxy S); TI OMAP3630 (Motorola Droid X); Apple A4 (iPad 1/iPhone 4). 2013: Allwinner A13, $5 in bulk.

Cryptographic details

The main NaCl work we did: achieve very high speeds without compromising security.

ECC, not RSA: much stronger security record.

Curve25519, not NSA/NIST curves: safecurves.cr.yp.to

Salsa20, not AES: much larger security margin.

Poly1305, not HMAC: information-theoretic security.

EdDSA, not ECDSA: collision-resilience et al.
3. Very fast rejection of forged packets under known public keys: no time spent on decryption. (This doesn't help much for forgeries under new keys, but flooded server can continue providing fast service to known keys.)

4. Fast batch verification, doubling speed of crypto_sign_open for valid signatures.

Also fast on small devices.

"NEON crypto" (CHES 2012) on 1GHz ARM Cortex-A8 core: 498349 cycles (2000/second) + 7.78 cycles/byte (1 Gbps) for box; and for verify: 624846 cycles (1600/second).

1GHz Cortex-A8 was high-end smartphone core in 2010: e.g., Samsung Exynos 3110 (Galaxy S); TI OMAP3630 (Motorola Droid X); Apple A4 (iPad 1/iPhone 4).

2013: Allwinner A13, $5 in bulk.

Cryptographic details

The main NaCl work we did: achieve very high speeds without compromising security.

ECC, not RSA: much stronger security record.

Curve25519, not NSA/NIST curves: safecurves.cr.yp.to

Salsa20, not AES: much larger security margin.

Poly1305, not HMAC: information-theoretic security.

EdDSA, not ECDSA: collision-resilience et al.
Also fast on small devices.

“NEON crypto” (CHES 2012) on 1GHz ARM Cortex-A8 core: 498349 cycles (2000/second) + 7.78 cycles/byte (1 Gbps) for box; and for verify: 624846 cycles (1600/second).

1GHz Cortex-A8 was high-end smartphone core in 2010: e.g., Samsung Exynos 3110 (Galaxy S); TI OMAP3630 (Motorola Droid X); Apple A4 (iPad 1/iPhone 4).

2013: Allwinner A13, $5 in bulk.

Cryptographic details

The main NaCl work we did: achieve very high speeds without compromising security.

ECC, not RSA: much stronger security record.

Curve25519, not NSA/NIST curves: safecurves.cr.yp.to

Salsa20, not AES: much larger security margin.

Poly1305, not HMAC: information-theoretic security.

EdDSA, not ECDSA: collision-resilience et al.
Also fast on small devices.

“NEON crypto” (CHES 2012) on 1GHz ARM Cortex-A8 core:
498349 cycles (2000/second)
+ 7.78 cycles/byte (1 Gbps)
for box; and for verify:
624846 cycles (1600/second).

1GHz Cortex-A8 was high-end smartphone core in 2010: e.g.,
Samsung Exynos 3110 (Galaxy S);
TI OMAP3630 (Motorola Droid X); Apple A4 (iPad 1/iPhone 4).

2013: Allwinner A13, $5 in bulk.

Cryptographic details

The main NaCl work we did: achieve very high speeds
without compromising security.

ECC, not RSA: much stronger security record.
Curve25519, not NSA/NIST curves: safecurves.cr.yp.to
Salsa20, not AES: much larger security margin.
Poly1305, not HMAC: information-theoretic security.
EdDSA, not ECDSA: collision-resilience et al.
Cryptographic details

The main NaCl work we did: achieve very high speeds \textit{without} compromising security.

ECC, not RSA: much stronger security record.

Curve25519, not NSA/NIST curves: \url{safecurves.cr.yp.to}

Salsa20, not AES: much larger security margin.

Poly1305, not HMAC: information-theoretic security.

EdDSA, not ECDSA: collision-resilience et al.

Case study: EdDSA

1985 ElGamal signatures: \((R, S)\) is signature of \(M\) if \(B^{H(M)} \equiv A^R R S \mod q\) and \(R, S \in \{0; 1; \ldots ; q-2\}\).

Here \(q\) is standard prime, \(B\) is standard base, \(A\) is signer's public key, \(H(M)\) is hash of message.

Signer generates \(A\) and \(R\) as secret powers of \(B\); easily solves for \(S\).
Cryptographic details

The main NaCl work we did: achieve very high speeds without compromising security.

ECC, not RSA: much stronger security record.
Curve25519, not NSA/NIST curves: safecurves.cr.yp.to
Salsa20, not AES: much larger security margin.
Poly1305, not HMAC: information-theoretic security.
EdDSA, not ECDSA: collision-resilience et al.

Case study: EdDSA

1985 ElGamal signatures $(R, S)$ is signature of $M$ if $B^{H(M)} \equiv A^R R^S \pmod{q}$ and $R, S \in \{0, 1, \ldots, q-1\}$.

Here $q$ is standard prime, $B$ is standard base, $A$ is signer’s public key, $H(M)$ is hash of message.

Signer generates $A$ and $R$ as secret powers of $B$; easily solves for $S$. 

1224.0x918.0
Cryptographic details

The main NaCl work we did:
achieve very high speeds
without compromising security.

ECC, not RSA:
much stronger security record.
Curve25519, not NSA/NIST
to curves: safecurves.cr.yp.to
Salsa20, not AES:
much larger security margin.
Poly1305, not HMAC:
information-theoretic security.
EdDSA, not ECDSA:
collision-resilience et al.

Case study: EdDSA

1985 ElGamal signatures:
(R, S) is signature of M
if \( B^{H(M)} \equiv A^R R^S \pmod{q} \)
and \( R, S \in \{0, 1, \ldots, q-2\} \).

Here \( q \) is standard prime,
\( B \) is standard base,
\( A \) is signer's public key,
\( H(M) \) is hash of message.

Signer generates \( A \) and \( R \)
as secret powers of \( B \);
easily solves for \( S \).
Cryptographic details
The main NaCl work we did: achieve very high speeds without compromising security.

ECC, not RSA: much stronger security record.
Curve25519, not NSA/NIST curves: safecurves.cr.yp.to
Salsa20, not AES: much larger security margin.
Poly1305, not HMAC: information-theoretic security.
EdDSA, not ECDSA: collision-resilience et al.

Case study: EdDSA
1985 ElGamal signatures: $(R, S)$ is signature of $M$ if $B^{H(M)} ≡ A^R R^S \pmod{q}$ and $R, S \in \{0, 1, \ldots, q − 2\}$.

Here $q$ is standard prime, $B$ is standard base, $A$ is signer’s public key, $H(M)$ is hash of message.

Signer generates $A$ and $R$ as secret powers of $B$; easily solves for $S$. 
Cryptographic details

In NaCl work we did:
- Achieve very high speeds
- Without compromising security.

Not RSA:
- Much stronger security record.

Not ECC, not NSA/NIST
- Curves: safecurves.cr.yp.to

Not AES:
- Much larger security margin.

Not Poly1305, not HMAC:
- Information-theoretic security.

Not EdDSA, not ECDSA:
- Collision-resilience et al.

Case study: EdDSA

1985 ElGamal signatures:

(R, S) is signature of M

if \( B^{H(M)} \equiv A^R R^S \pmod q \)

and \( R, S \in \{0, 1, \ldots, q - 2\} \).

Here \( q \) is standard prime,
\( B \) is standard base,
\( A \) is signer’s public key,
\( H(M) \) is hash of message.

Signer generates \( A \) and \( R \)
as secret powers of \( B \);
easily solves for \( S \).

1990 Schnorr improvements:

1. Hash \( R \) in the exponent:
   \( B^{H(M)} \equiv A^{R^R} R^S \).
   Reduces attacker control.

2. Replace three exponents
   with two exponents:
   \( B^{H(M)} = H^R(R) \equiv A^{R^R} = H^R(R) \).
   Saves time in verification.

3. Simplify by relabeling \( S \):
   \( B^{H(M)} = H^R(R) \equiv A^{R^R} \).
   Saves time in verification.

4. Merge the hashes:
   \( B^{H(R,M)} \equiv A^{R^R} \).
   \( \Rightarrow \) Resilient to \( H \) collisions.
The main NaCl work we did:
achieve very high speeds
without compromising security.

ECC, not RSA:
much stronger security record.
Curve25519, not NSA/NIST
curves:
safecurves.cr.yp.to
Salsa20, not AES:
much larger security margin.
Poly1305, not HMAC:
information-theoretic security.
EdDSA, not ECDSA:
collision-resilience et al.

Case study: EdDSA
1985 ElGamal signatures:
\((R, S)\) is signature of \(M\)
if \(B^{H(M)} \equiv A^R R^S \pmod{q}\)
and \(R, S \in \{0, 1, \ldots, q-2\}\).

Here \(q\) is standard prime,
\(B\) is standard base,
\(A\) is signer’s public key,
\(H(M)\) is hash of message.

Signer generates \(A\) and \(R\)
as secret powers of \(B\);
easily solves for \(S\).

1990 Schnorr improvements:
1. Hash \(R\) in the exponent:
   \(B^{H(M)} \equiv A^{H(R)} R^S\).
   Reduces attacker control.
2. Replace three exponents
   with two exponents:
   \(B^{H(M)}/H(R) \equiv A R^S\).
   Saves time in verification.
3. Simplify by relabeling \(S\):
   \(B^{H(M)}/H(R) \equiv A R^S\).
   Saves time in verification.
4. Merge the hashes:
   \(B^{H(R,M)} \equiv AR^S\).
   \(\Rightarrow\) Resilient to \(H\) collisions.
Case study: EdDSA

1985 ElGamal signatures:

\((R, S)\) is signature of \(M\)

if \(B^{H(M)} \equiv A^R R^S \pmod{q}\)

and \(R, S \in \{0, 1, \ldots, q - 2\}\).

Here \(q\) is standard prime,

\(B\) is standard base,

\(A\) is signer's public key,

\(H(M)\) is hash of message.

Signer generates \(A\) and \(R\)

as secret powers of \(B\);

easily solves for \(S\).

1990 Schnorr improvements:

1. Hash \(R\) in the exponent:

\(B^{H(M)} \equiv A^{H(R)} R^S\).

Reduces attacker control.

2. Replace three exponents with two exponents:

\(B^{H(M)/H(R)} \equiv A^{R^S/H(R)}\).

Saves time in verification.

3. Simplify by relabeling \(S\):

\(B^{H(M)/H(R)} \equiv A^{R^S}\).

Saves time in verification.

4. Merge the hashes:

\(B^{H(R,M)} \equiv A^{R^S}\).

\(\Rightarrow\) Resilient to \(H\) collisions.
Case study: EdDSA

1985 ElGamal signatures:

\((R, S)\) is signature of \(M\)

if \(B^{H(M)} \equiv A^R R^S \pmod{q}\)

and \(R, S \in \{0, 1, \ldots, q - 2\}\).

Here \(q\) is standard prime,

\(B\) is standard base,

\(A\) is signer’s public key,

\(H(M)\) is hash of message.

Signer generates \(A\) and \(R\)
as secret powers of \(B\);
easily solves for \(S\).

1990 Schnorr improvements:

1. Hash \(R\) in the exponent:

\(B^{H(M)} \equiv A^{H(R)} R^S\).

Reduces attacker control.

2. Replace three exponents

with two exponents:

\(B^{H(M)/H(R)} \equiv A^{R^S/H(R)}\).

Saves time in verification.

3. Simplify by relabeling \(S\):

\(B^{H(M)/H(R)} \equiv A^{R^S}\).

Saves time in verification.

4. Merge the hashes:

\(B^{H(R,M)} \equiv A^{R^S}\).

⇒ Resilient to \(H\) collisions.
1990 Schnorr improvements:

1. Hash $R$ in the exponent:
   
   $$B^{H(M)} \equiv A^{H(R)} R^S.$$

   Reduces attacker control.

2. Replace three exponents with two exponents:
   
   $$B^{H(M)/H(R)} \equiv AR^{S/H(R)}.$$

   Saves time in verification.

3. Simplify by relabeling $S$:
   
   $$B^{H(M)/H(R)} \equiv AR^S.$$

   Saves time in verification.

4. Merge the hashes:
   
   $$B^{H(R,M)} \equiv AR^S.$$

   $\Rightarrow$ Resilient to $H$ collisions.

5. Eliminate inversions for signer:
   
   $$B^S \equiv R^S.$$

   Simpler, faster.

6. Compress $R$ to $H(R;M)$.

   Saves space in signatures.

7. Use half-size $H$ output.

   Saves space in signatures.
1985 ElGamal signatures:

\((R; S)\) is signature of \(M\) if

\[ B H(M) \equiv A R R S \pmod{q} \]

and \(R; S \in \{0, 1, \ldots, q - 2\}\).

Here \(q\) is standard prime, \(B\) is standard base, \(A\) is signer's public key, \(H(M)\) is hash of message.

Signer generates \(A\) and \(R\) as secret powers of \(B\); easily solves for \(S\).

1990 Schnorr improvements:

1. Hash \(R\) in the exponent:

\[ B^{H(M)} \equiv A^{H(R)} R S. \]

Reduces attacker control.

2. Replace three exponents with two exponents:

\[ B^{H(M)/H(R)} \equiv A R S^{H(R)}. \]

Saves time in verification.

3. Simplify by relabeling \(S\):

\[ B^{H(M)/H(R)} \equiv A R S. \]

Saves time in verification.

4. Merge the hashes:

\[ B^{H(R,M)} \equiv A R S. \]

\(\Rightarrow\) Resilient to \(H\) collisions.

5. Eliminate inversions for signer:

\[ B^S \equiv R A^{H(R,M)}. \]

Simpler, faster.

6. Compress \(R\) to \(H(R;M)\).

Saves space in signatures.

7. Use half-size \(H\) output.

Saves space in signatures.
1990 Schnorr improvements:
1. Hash $R$ in the exponent:
   $B^{H(M)} \equiv A^{H(R)} R^S$.
   Reduces attacker control.
2. Replace three exponents with two exponents:
   $B^{H(M)/H(R)} \equiv A R^{S/H(R)}$.
   Saves time in verification.
3. Simplify by relabeling $S$:
   $B^{H(M)/H(R)} \equiv A R^S$.
   Saves time in verification.
4. Merge the hashes:
   $B^{H(R,M)} \equiv A R^S$.
   $\Rightarrow$ Resilient to $H$ collisions.
5. Eliminate inversions for signer:
   $B^S \equiv R A^{H(R,M)}$.
   Simpler, faster.
6. Compress $R$ to $H(R,M)$.
   Saves space in signatures.
7. Use half-size $H$ output.
   Saves space in signatures.
1990 Schnorr improvements:

1. Hash \( R \) in the exponent:
\[ B^{H(M)} \equiv A^{H(R)} R^S. \]
Reduces attacker control.

2. Replace three exponents with two exponents:
\[ B^{H(M)/H(R)} \equiv A R^{S/H(R)}. \]
Saves time in verification.

3. Simplify by relabeling \( S \):
\[ B^{H(M)/H(R)} \equiv A R^S. \]
Saves time in verification.

4. Merge the hashes:
\[ B^{H(R,M)} \equiv A R^S. \]
\[ \Rightarrow \text{Resilient to } H \text{ collisions}. \]

5. Eliminate inversions for signer:
\[ B^S \equiv R A^{H(R,M)}. \]
Simpler, faster.

6. Compress \( R \) to \( H(R, M) \).
Saves space in signatures.

7. Use half-size \( H \) output.
Saves space in signatures.
1990 Schnorr improvements:

1. Hash $R$ in the exponent:
   \[ B^{H(M)} \equiv A^{H(R)} R^S. \]
   Reduces attacker control.

2. Replace three exponents with two exponents:
   \[ B^{H(M)/H(R)} \equiv A^{R^S/H(R)}. \]
   Saves time in verification.

3. Simplify by relabeling $S$:
   \[ B^{H(M)/H(R)} \equiv A^{R^S}. \]
   Saves time in verification.

4. Merge the hashes:
   \[ B^{H(R,M)} \equiv A^{R^S}. \]
   ⇒ Resilient to $H$ collisions.

5. Eliminate inversions for signer:
   \[ B^S \equiv RA^{H(R,M)}. \]
   Simpler, faster.

6. Compress $R$ to $H(R, M)$.
   Saves space in signatures.

7. Use half-size $H$ output.
   Saves space in signatures.

Subsequent research:
Extensive theoretical study of security of Schnorr’s system.
1990 Schnorr improvements:

1. Hash $R$ in the exponent:
   \[ B^{H(M)} \equiv A^{H(R)} R^S. \]
   Reduces attacker control.

2. Replace three exponents with two exponents:
   \[ B^{H(M)/H(R)} \equiv A^{R S/H(R)}. \]
   Saves time in verification.

3. Simplify by relabeling $S$:
   \[ B^{H(M)/H(R)} \equiv A R^S. \]
   Saves time in verification.

4. Merge the hashes:
   \[ B^{H(R, M)} \equiv A R^S. \]
   \[ \Rightarrow \text{Resilient to } H \text{ collisions.} \]

5. Eliminate inversions for signer:
   \[ B^S \equiv R A^{H(R, M)}. \]
   Simpler, faster.

6. Compress $R$ to $H(R, M)$.
   Saves space in signatures.

7. Use half-size $H$ output.
   Saves space in signatures.

Subsequent research:

extensive theoretical study of security of Schnorr’s system.

But patented. \[ \Rightarrow \text{DSA, ECDSA avoided most improvements.} \]
1990 Schnorr improvements:

1. Hash $R$ in the exponent:
   \[ B^{H(M)} \equiv A^{H(R)} R^S. \]
   Reduces attacker control.

2. Replace three exponents with two exponents:
   \[ B^{H(M)} / H(R) \equiv A R^{S/H(R)}. \]
   Saves time in verification.

3. Simplify by relabeling $S$:
   \[ B^{H(M)} / H(R) \equiv A R^S. \]
   Saves time in verification.

4. Merge the hashes:
   \[ B^{H(R,M)} \equiv A R^S. \]
   ⇒ Resilient to $H$ collisions.

5. Eliminate inversions for signer:
   \[ B^S \equiv R A^{H(R,M)}. \]
   Simpler, faster.

6. Compress $R$ to $H(R, M)$.
   Saves space in signatures.

7. Use half-size $H$ output.
   Saves space in signatures.

Subsequent research:

- extensive theoretical study of security of Schnorr’s system.
  But patented. ⇒ DSA, ECDSA avoided most improvements.

1990 Schnorr improvements:

1. Hash $R$ in the exponent:
$$B_H(M) \equiv A_H(R)R.$$

Reduces attacker control.

2. Replace three exponents with two exponents:
$$B_H(M) = H(R) \equiv ARS = H(R)S.$$

Saves time in verification.

3. Simplify by relabeling $S$:
$$B_H(M) = H(R) \equiv ARS.$$

Saves time in verification.

4. Merge the hashes:
$$B_H(R;M) \equiv ARS.$$

⇒ Resilient to $H$ collisions.

5. Eliminate inversions for signer:
$$BS \equiv RA_H(R;M).$$

Simpler, faster.

6. Compress $R$ to $H(R;M)$.

Saves space in signatures.

7. Use half-size $H$ output.

Saves space in signatures.

Subsequent research:

But patented. $\Rightarrow$ DSA, ECDSA avoided most improvements.

Generative security of Schnorr's system.

Extensive theoretical study:

Avoid PlayStation disaster.


Use elliptic curves in "complete $−1$-twisted Edwards" form.

⇒ Very high speed, natural side-channel protection, no exceptional cases.

Skip signature compression.

Support batch verification.

Use double-size $H$ output, and include $A$ as input.

Generate $R$ deterministically as a secret hash of $M$.

⇒ Avoid PlayStation disaster.
1990 Schnorr improvements:

1. Hash \( R \) in the exponent:
   \[
   B H(M) \equiv A H(R) R S.
   \]
   Reduces attacker control.

2. Replace three exponents with two exponents:
   \[
   B H(M) = H(R) \equiv AR S = H(R) S.
   \]
   Saves time in verification.

3. Simplify by relabeling \( S \):
   \[
   B H(M) = H(R) \equiv AR S.
   \]
   Saves time in verification.

4. Merge the hashes:
   \[
   B H(R;M) \equiv AR S.
   \]
   \( \Rightarrow \) Resilient to \( H \) collisions.

5. Eliminate inversions for signer:
   \[
   B^S \equiv RA^{H(R,M)}.
   \]
   Simpler, faster.

6. Compress \( R \) to \( H(R, M) \).
   Saves space in signatures.

7. Use half-size \( H \) output.
   Saves space in signatures.

Subsequent research:

extensive theoretical study of security of Schnorr’s system.

But patented. \( \Rightarrow \) DSA, ECDSA avoided most improvements.


Use elliptic curves in “complete − 1-twisted Edwards” form.

⇒ very high speed, natural side-channel protection, no exceptional cases.

Skip signature compression.

Support batch verification.

Use double-size \( H \) output, and include \( A \) as input.

Generate \( R \) deterministically as a secret hash of \( M \).

⇒ Avoid PlayStation disaster.
5. Eliminate inversions for signer:
\[ B^S \equiv RA^{H(R,M)}. \]
Simpler, faster.

6. Compress \( R \) to \( H(R, M) \).
Saves space in signatures.

7. Use half-size \( H \) output.
Saves space in signatures.

Subsequent research:
extensive theoretical study of security of Schnorr’s system.

But patented. \( \Rightarrow \) DSA, ECDSA avoided most improvements.


Use elliptic curves in “complete $-1$-twisted Edwards” form.
\( \Rightarrow \) very high speed, natural side-channel protection, no exceptional cases.

Skip signature compression.
Support batch verification.

Use double-size \( H \) output, and include \( A \) as input.

Generate \( R \) deterministically as a secret hash of \( M \).
\( \Rightarrow \) Avoid PlayStation disaster.
5. Eliminate inversions for signer:
\[ B^S \equiv RA^{H(R,M)}. \]
Simpler, faster.

6. Compress \( R \) to \( H(R, M) \).
Saves space in signatures.

7. Use half-size \( H \) output.
Saves space in signatures.

Subsequent research:
extensive theoretical study of
security of Schnorr’s system.

But patented. \( \Rightarrow \) DSA, ECDSA
avoided most improvements.


Use elliptic curves in “complete
\(-1\)-twisted Edwards” form.
\( \Rightarrow \) very high speed,
natural side-channel protection,
no exceptional cases.

Skip signature compression.
Support batch verification.

Use double-size \( H \) output,
and include \( A \) as input.

Generate \( R \) deterministically
as a secret hash of \( M \).
\( \Rightarrow \) Avoid PlayStation disaster.