

Cleaning up crypto

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

Tanja Lange, T. U. Eindhoven

Joint work with:

Peter Schwabe, R. U. Nijmegen

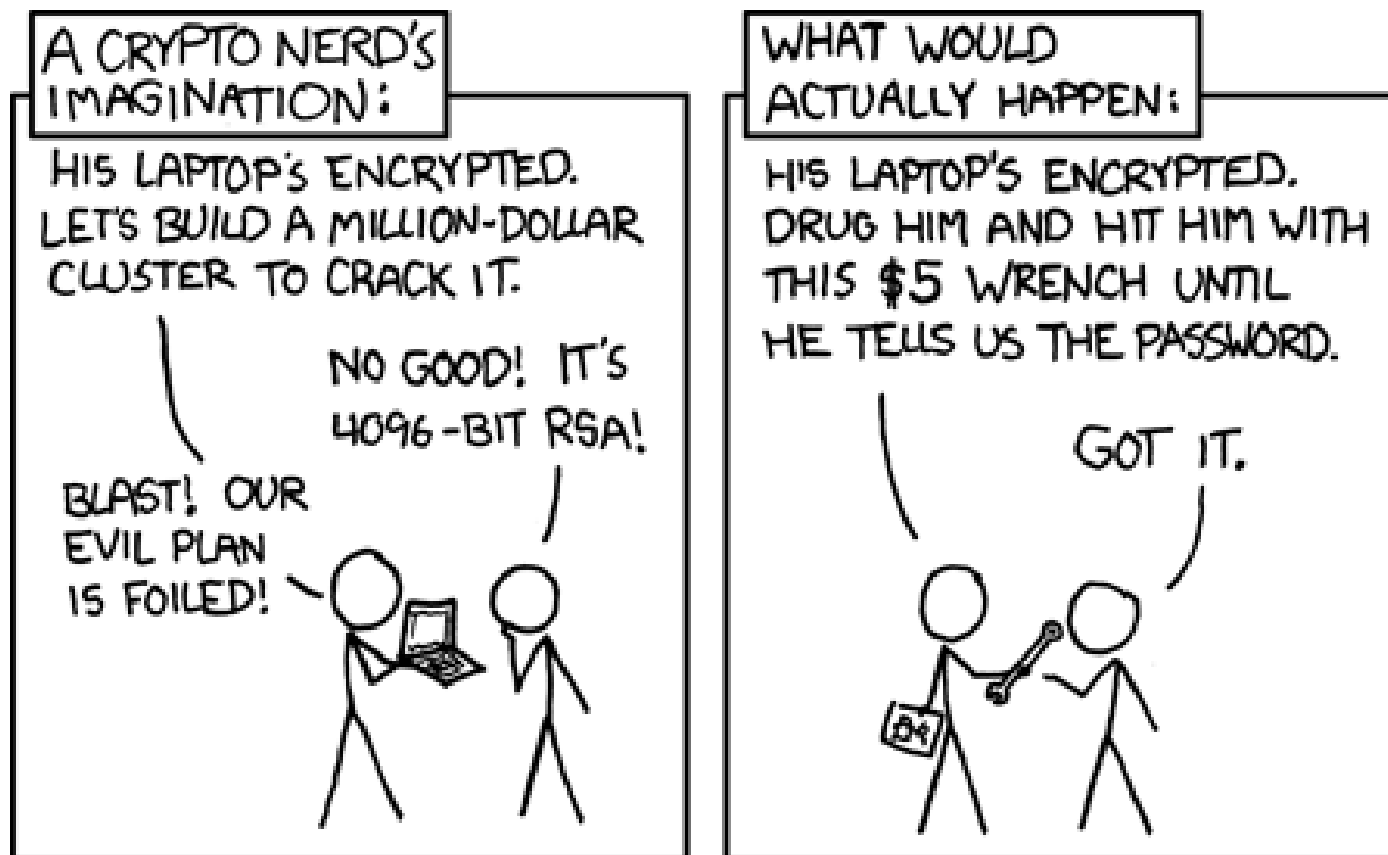
AES-128, RSA-2048, etc.

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.



<http://xkcd.com/538/>

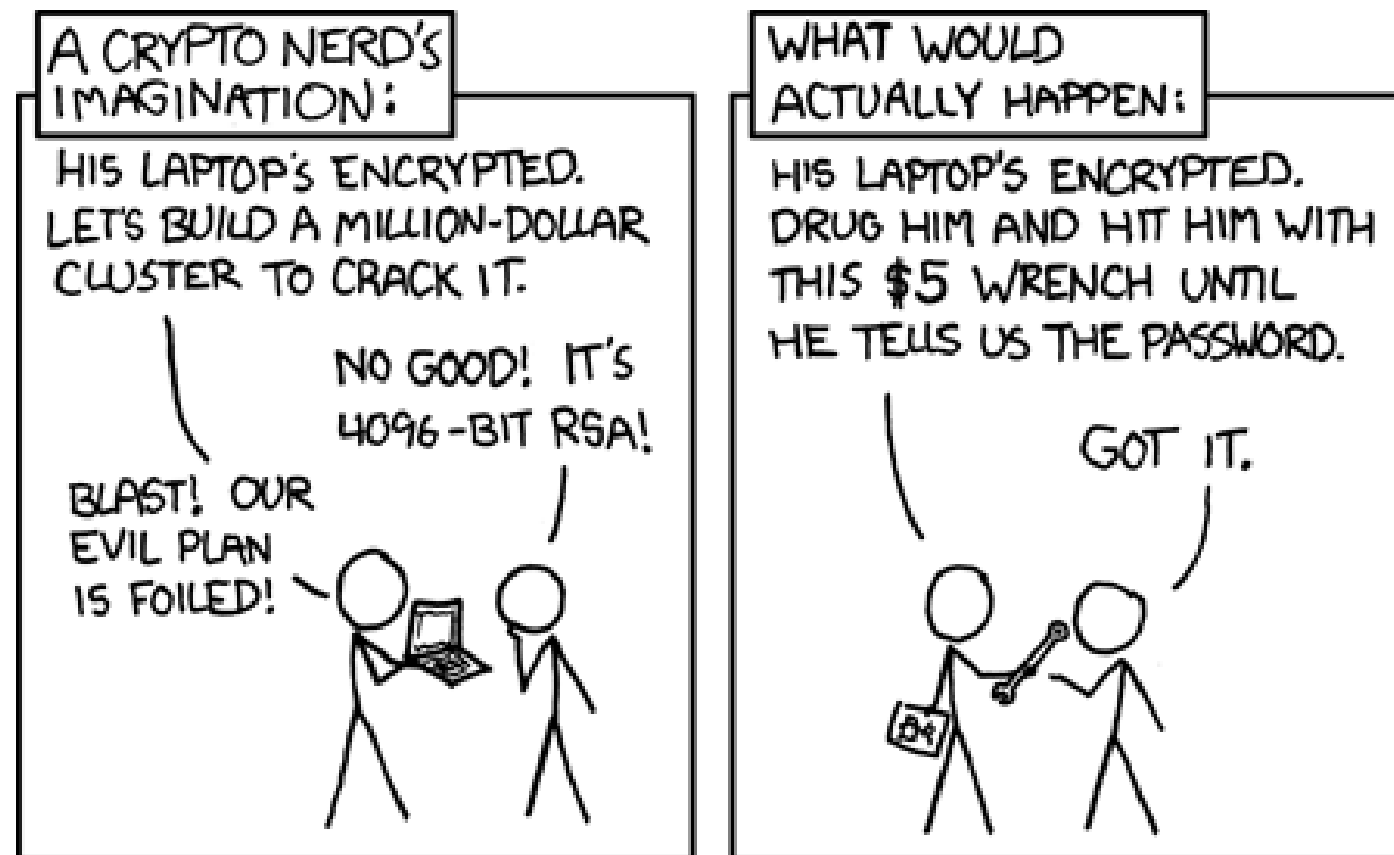
Cleaning up crypto

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

Tanja Lange, T. U. Eindhoven

Joint work with:

Peter Schwabe, R. U. Nijmegen



<http://xkcd.com/538/>

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.

g up crypto

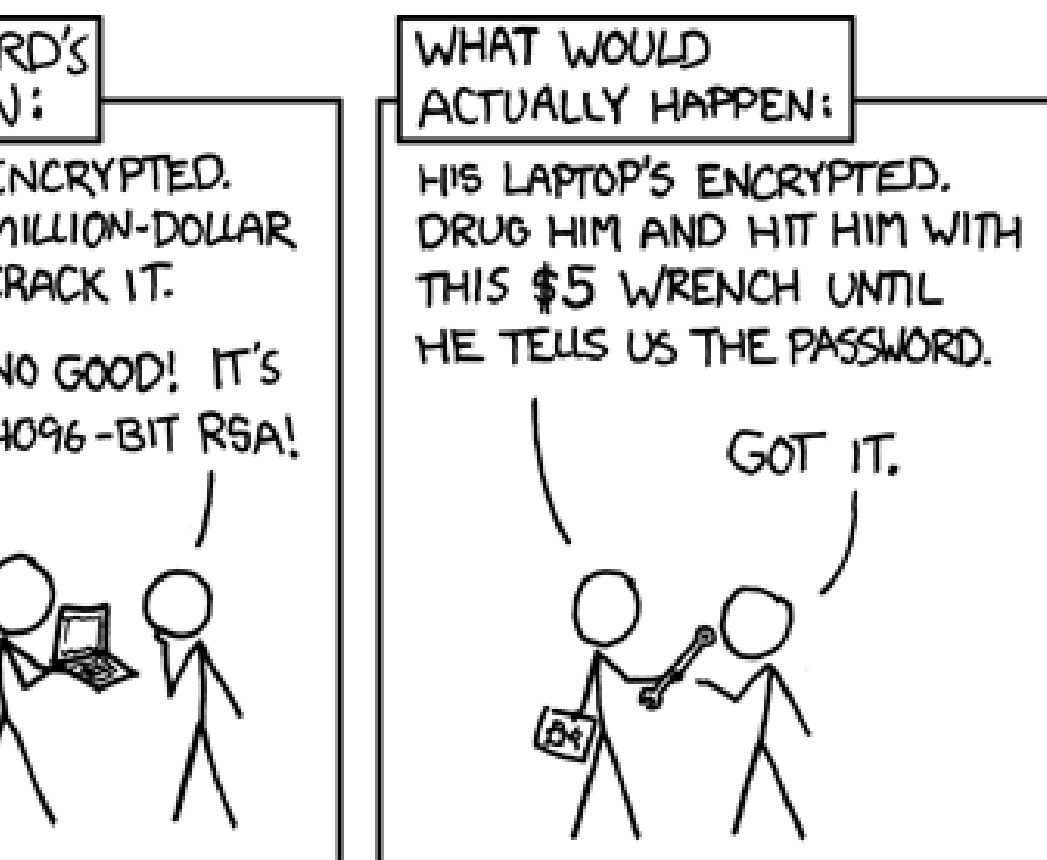
ernstein, U. Illinois Chicago

Eindhoven

ange, T. U. Eindhoven

ork with:

chwabe, R. U. Nijmegen



[/xkcd.com/538/](http://xkcd.com/538/)

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

a new cr

NaCl ("s

the unde

[nacl.ca](http://nacl.cr)

nacl.cr

and exte

Acknowl

code cor

Matthew

Media),

Emilia K

Adam L

Bo-Yin Y

o
. Illinois Chicago
n
J. Eindhoven
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
a new cryptographic
NaCl ("salt"), to
the underlying pro

nacl.cace-project.org
nacl.cr.yp.to:
and extensive docu

Acknowledgments:
code contributions
Matthew Dempsy
Media), Niels Duif
Emilia Käsper (Le
Adam Langley (Go
Bo-Yin Yang (Aca



m/538/

Chicago

en

gen



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+impleme
a new cryptographic library,
NaCl ("salt"), to address
the underlying problems.

nacl.cace-project.eu,
nacl.cr.yt.to: source
and extensive documentation

Acknowledgments:
code contributions from
Matthew Dempsy (Mochi
Media), Niels Duif (Eindhov
Emilia Käsper (Leuven),
Adam Langley (Google),
Bo-Yin Yang (Academia Sin

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.cace-project.eu,
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).

3, RSA-2048, etc.

ly accepted standards.

ly infeasible to break

attacks in literature.

entations are available

c cryptographic libraries

OpenSSL.

n security practice is

nose implementations.

otography is still

er! Complete failures

identiality and integrity.

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

nacl.cace-project.eu,

nacl.cr.yp.to: source

and extensive documentation.

Acknowledgments:

code contributions from

Matthew Dempsy (Mochi

Media), Niels Duif (Eindhoven),

Emilia Käsper (Leuven),

Adam Langley (Google),

Bo-Yin Yang (Academia Sinica).

Most of

is crypto

Primary

Main tas

authent

Alice has

Uses Bo

Alice’s s

authenti

Sends c

Bob uses

and Bob

to verify

48, etc.
d standards.
le to break
literature.
re available
aphic libraries
practice is
ementations.
is still
ete failures
and integrity.

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

nacl.cace-project.eu,
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 Intern
is cryptographically
Primary goal of Na
Main task: **public**
authenticated en
Alice has a messag
Uses Bob’s public
Alice’s secret key t
authenticated ciph
Sends c to Bob.
Bob uses Alice’s p
and Bob’s secret k
to verify and recov

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

nacl.cace-project.eu,
nacl.cr.yt.to: source and extensive documentation.

Acknowledgments:
code contributions from
Matthew Dempsy (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 that.

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+implemented a new cryptographic library, NaCl (“salt”), to address the underlying problems.

nacl.cace-project.eu,
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 .

e designed+implemented
cryptographic library,
"salt"), to address
underlying problems.

[nacl-cr.yp.to](https://nacl.cr.yp.to),
source
comprehensive documentation.

acknowledgments:

contributions from
David Dempsy (Mochi
Niels Duif (Eindhoven),
Kasper (Leuven),
Langley (Google),
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 uses
typical operations

Generate
Use AES

Hash encryption
Read RSA

Use key
Read Bob's

Use key
Convert

Plus more
allocate

handle encryption

+implemented
nic library,
address
blems.
ect . eu,
source
umentation.
s from
y (Mochi
F (Eindhoven),
uven),
oogle),
ademia 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 cryptograp
Generate random
Use AES key to en
Hash encrypted pa
Read RSA key from
Use key to sign ha
Read Bob's key fro
Use key to encrypt
Convert to wire fo
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 for

Use key to sign hash.

Read Bob's key from wire for

Use key to encrypt signature

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.

the Internet
graphically unprotected.
goal of NaCl: Fix this.

ask: **public-key
authenticated encryption.**

has a message m for Bob.

Bob's public key and
secret key to compute
authenticated ciphertext c .
Send c to Bob.

Bob has Alice's public key
and Alice's secret key
to decrypt c 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
 $c = \text{crypt}$

packet
is unprotected.
NaCl: Fix this.

-key
encryption.

message m for Bob.

key and
to compute
ciphertext c .

public key
key
over 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 = crypto_box(m,`

ected.
his.

Bob.

te

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, s)
```

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)
```

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)
```

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.

ng a
cryptographic library:
e random AES key.
S key to encrypt packet.
rypted packet.
SA key from wire format.
to sign hash.
b's key from wire format.
to encrypt signature etc.
to wire format.

re code:
storage,
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 veri

`m=crypt`

Initial ke

`pk = cry`

chic library:

AES key.

ncrypt packet.

acket.

m wire format.

sh.

om wire format.

t signature etc.

rmat.

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, dec

```
m=crypto_box_op
```

Initial key generati

```
pk = crypto_box.
```

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, sk = 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)
```

ng NaCl:

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

secret key sk.

public key pk.

nonce n.

bytes longer than m.

cts are C++

tring variables

ted in wire format,

r storage/transmission.

similar, using pointers;

ory 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 so

Don’t ap

m, n, pk, sk)

sk.

pk.

er than m.

+

ables

e format,

transmission.

sing pointers;

tion, 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

Don't applications

k)

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)
```

on.

ers;

ilures.

“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?”

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.

ifying, decrypting:

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

ey generation:

```
crypto_box_keypair(&sk)
```

ead use **signatures**

c messages:

```
crypto_sign_keypair(&sk)
```

secret key,

public key.

```
crypto_sign(m, sk)
```

overhead.

```
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.

No secrets

2005 Os

65ms to

used for

Attack p

but with

Almost a

use fast

Kernel’s

influence

influenci

influenci

of the at

65ms to

encrypting:
open(c, n, pk, sk)

ion:
_keypair(&sk)

Signatures

es:
n_keypair(&sk)

n(m, sk)

_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.

No secret load add

2005 Osvik–Shamir
65ms to steal Linux
used for hard-disk
Attack process on
but without privile

Almost all AES im
use fast lookup ta
Kernel’s secret AE
influences table-lo
influencing CPU c
influencing measur
of the attack proc
65ms to compute

pk, sk)

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

(&sk)

Examples of applications
using NaCl’s crypto_box:

r(&sk)

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.

, pk)

Threema, encrypted-chat app.

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 implementat
use fast lookup tables.

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

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.

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.

sounds too simple!

applications need more?"

es of applications

aCI's crypto_box:

ve and DNSCrypt,

urity authenticated

on for DNS queries;

l by OpenDNS.

Google's TLS replacement.

LT in Ethos OS,

LS replacement.

a, encrypted-chat app.

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 cry

still use

but add

intended

upon the

Not con

likely to

simple!
need more?"
ications
to_box:
NSCrypt,
enticated
S queries;
DNS.
LS replacement.
s OS,
ement.
ed-chat app.

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
still use secret load
but add "counterm
intended to obscur
upon the CPU cac
Not confidence-ins
likely to be breaka

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.

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.

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.

2010 Langley ctgrind:
verify this automatically.

secret load addresses

Barak–Shamir–Tromer:
steal Linux AES key
hard-disk encryption.
process on same CPU
without privileges.

all AES implementations
lookup tables.

secret AES key
uses table-load addresses,
changing CPU cache state,
changing measurable timings
during attack process.
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.

2010 Langley ctgrind:
verify this automatically.

No secret

2011 Br
minutes
machine
Secret b
influence

Most cry
has man
variation
e.g., mem

addresses

ir–Tromer:
ix AES key
encryption.
same CPU
ages.

implementations
bles.

S key
ad addresses,
ache state,
rable timings
ess.

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.

2010 Langley ctgrind:
verify this automatically.

No secret branch c

2011 Brumley–Tuv
minutes to steal a
machine’s OpenSS
Secret branch con
influence timings.

Most cryptographic
has many more sm
variations in timin
e.g., memcmp for IP

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.

2010 Langley ctgrind: verify this automatically.

No secret branch conditions

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

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

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.

2010 Langley ctgrind:
verify this automatically.

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.

2010 Langley ctgrind: verify this automatically.

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.

cryptographic libraries
secret load addresses
“countermeasures”
to obscure influence
CPU cache state.
confidence-inspiring;
be breakable.

systematically avoids
s from addresses
depend on secret data.
es this type of disaster.

ngley ctgrind:
is automatically.

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 padd

1998 Ble
Decrypt
by obser
to $\approx 10^6$

SSL first
then che
(which r
Subsequ
more ser

Server re
pattern
pattern

ic libraries
d addresses
measures”
re influence
che state.
spiring;
ble.
ly avoids
resses
cret data.
e of disaster.
rind:
tically.

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 oracle

1998 Bleichenbach
Decrypt SSL RSA
by observing server
to $\approx 10^6$ variants of
SSL first inverts R
then checks for “F
(which many forge
Subsequent proces
more serious integ
Server responses r
pattern of PKCS f
pattern reveals pla

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 response
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 $\approx 10^6$ 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.

Secret branch conditions

Humley–Tuveri:

to steal another

's OpenSSL ECDSA key.

branch conditions

the timings.

Cryptographic software

by more small-scale

as in timing:

icmp for IPsec MACs.

systematically avoids

such conditions

depend on secret data.

prevents 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 c

try to hi

between

subsequ

But hard

e.g., Cry

Focardi–

2013.02.

conditions

veri:
another
SSL ECDSA key.
ditions

ic software
small-scale
g:
Psec MACs.

ly avoids
ons
cret data.
e 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 str
try to hide differen
between padding o
subsequent integri
But hard to get th
e.g., Crypto 2012
Focardi–Kawamoto
2013.02.04 Alfarda

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: s
e.g., Crypto 2012 Bardou–
Focardi–Kawamoto–Steel–T
2013.02.04 Alfardan–Paterso

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., Crypto 2012 Bardou–
Focardi–Kawamoto–Steel–Tsay,
2013.02.04 Alfardan–Paterson.

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., Crypto 2012 Bardou–
Focardi–Kawamoto–Steel–Tsay,
2013.02.04 Alfardan–Paterson.

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.

Timing oracles

Reichenbacher:

SSL RSA ciphertext
Timing server responses
variants of ciphertext.

... inverts RSA,
checks for “PKCS padding”
(many forgeries have).
... processing applies
various integrity checks.
... responses reveal
... of PKCS forgeries;
... 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., Crypto 2012 Bardou–
Focardi–Kawamoto–Steel–Tsay,
2013.02.04 Alfardan–Paterson.

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.

Centraliz

2008 Be
OpenSS
had only
Debian c
a subtle
randomr

s
mer:
ciphertext
r responses
of ciphertext.
SA,
PKCS padding”
eries have).
ssing applies
rity checks.
eveal
forgeries;
plaintext.

Typical defense strategy:
try to hide differences
between padding checks and
subsequent integrity checks.
But hard to get this right: see,
e.g., Crypto 2012 Bardou–
Focardi–Kawamoto–Steel–Tsay,
2013.02.04 Alfardan–Paterson.
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 random
2008 Bello: Debian
OpenSSL keys for
had only 15 bits o
Debian developer
a subtle line of Op
randomness-genera

Typical defense strategy:

try to hide differences
between padding checks and
subsequent integrity checks.

But hard to get this right: see,
e.g., Crypto 2012 Bardou–
Focardi–Kawamoto–Steel–Tsay,
2013.02.04 Alfardan–Paterson.

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., Crypto 2012 Bardou–
Focardi–Kawamoto–Steel–Tsay,
2013.02.04 Alfardan–Paterson.

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., Crypto 2012 Bardou–
Focardi–Kawamoto–Steel–Tsay,
2013.02.04 Alfardan–Paterson.

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.

defense strategy:

code differences

padding checks and
content integrity checks.

needed to get this right: see,

crypto 2012 Bardou–

–Kawamoto–Steel–Tsay,

2004 Alfardan–Paterson.

messages not decrypt

if message is authenticated.

Verification procedure rejects

messages in constant time.

These are further constrained

by the once 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

merge m

pool feed

Merging

auditabl

bad/faili

if there i

strategy:
ances
checks and
ty checks.
his right: see,
Bardou–
o–Steel–Tsay,
an–Paterson.
crypt
authenticated.
dure rejects
stant time.
r constrained
separation
ce 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 allo
merge many entrop
pool feeding many
Merging is determ
auditable. Can sur
bad/failing/malicio
if there is one goo

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
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

Example: Debian/Ubuntu

Generate keys for 1.5 years

with 15 bits of entropy.

A developer had removed

a line of OpenSSL

business-generating code.

uses `/dev/urandom`,

a random-number generator.

Moving this kernel code

is more tractable than

having separate RNG code

in a security library.

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

2010 Bu

Sven: S

requirem

for each

leaked P

mnness

n/Ubuntu

1.5 years

f entropy.

had removed

OpenSSL

ating code.

random,

umber generator.

nel code

table than

e RNG code

brary.

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 unnecess

2010 Bushing–Ma

Sven: Sony ignore

requirement of new

for each signature.

leaked PS3 code-s

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

2010 Bushing–Marcan–Segher

Sven: Sony ignored ECDSA

requirement of new randomness

for each signature. \Rightarrow Signature

leaked PS3 code-signing key

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

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

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

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 eBACS (ECRYPT Benchmarking of Cryptographic Systems).

ization allows OS to
many entropy sources into
ding many applications.

is deterministic and
e. Can survive many
ng/malicious sources
is one good source.

ep backwards:

DRAND in applications.

entropy source; no backup;

be poorly cloned;

rable (CHES 2013);

itable. Not used in NaCl.

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
eBACS (ECRYPT Benchmarking
of Cryptographic Systems).

Avoiding

2008 Ste

Appelba

Osvik–d

MD5 \Rightarrow

ws OS to
py sources into
y applications.

inistic and
rvive many
ous sources
d source.

nds:
n applications.
rce; no backup;
cloned;
ES 2013);
t used in NaCl.

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
eBACS (ECRYPT Benchmarking
of Cryptographic Systems).

Avoiding pure cryp

2008 Stevens–Soti
Appelbaum–Lenstr
Osvik–de Weger e
MD5 \Rightarrow rogue CA

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 eBACS (ECRYPT Benchmarking of Cryptographic Systems).

Avoiding pure crypto failures

2008 Stevens–Sotirov–Appelbaum–Lenstra–Molnar–Osvik–de Weger exploited MD5 \Rightarrow rogue CA cert.

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 eBACS (ECRYPT Benchmarking of Cryptographic Systems).

Avoiding pure crypto failures

2008 Stevens–Sotirov–Appelbaum–Lenstra–Molnar–Osvik–de Weger exploited MD5 \Rightarrow rogue CA cert.

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 eBACS (ECRYPT Benchmarking of Cryptographic Systems).

Avoiding pure crypto failures

2008 Stevens–Sotirov–Appelbaum–Lenstra–Molnar–Osvik–de Weger exploited MD5 \Rightarrow rogue CA cert.
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 eBACS (ECRYPT Benchmarking of Cryptographic Systems).

Avoiding pure crypto failures

2008 Stevens–Sotirov–Appelbaum–Lenstra–Molnar–Osvik–de Weger exploited MD5 \Rightarrow rogue CA cert.
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.

g unnecessary randomness

shing–Marcan–Segher–

ony ignored ECDSA

ment of new randomness

signature. \Rightarrow Signatures

PS3 code-signing key.

s deterministic

_box and `crypto_sign`.

ness only for `keypair`.

es this type of disaster.

plifies testing. NaCl uses

ed test battery from

(ECRYPT Benchmarking

ographic Systems).

Avoiding pure crypto failures

2008 Stevens–Sotirov–

Appelbaum–Lenstra–Molnar–

Osvik–de Weger exploited

MD5 \Rightarrow rogue CA cert.

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 p

often lea

cryptogr

or give u

Example

used RS

Security

Analyses

that RSA

e.g., 200

estimate

RSA Lab

Move to

ary randomness

rcan–Segher–
d ECDSA

w randomness

. ⇒ Signatures
igning key.

histic

crypto_sign.
for keypair.
e of disaster.

ting. NaCl uses
ttery from
Benchmarking
Systems).

Avoiding pure crypto failures

2008 Stevens–Sotirov–
Appelbaum–Lenstra–Molnar–
Osvik–de Weger exploited
MD5 ⇒ rogue CA cert.

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
often lead users to
cryptographic secu
or give up on cryp

Example 1: Google
used RSA-1024 un

Security note:
Analyses in 2003 c
that RSA-1024 wa
e.g., 2003 Shamir–
estimated 1 year, a
RSA Labs and NIS
Move to RSA-204

Business

er-

ness

atures

.

ign.

ir.

ter.

l uses

n

rking

Avoiding pure crypto failures

2008 Stevens–Sotirov–
Appelbaum–Lenstra–Molnar–

Osvik–de Weger exploited

MD5 \Rightarrow rogue CA cert.

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

2008 Stevens–Sotirov–
Appelbaum–Lenstra–Molnar–

Osvik–de Weger exploited

MD5 \Rightarrow rogue CA cert.

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.

g pure crypto failures

evens–Sotirov–

um–Lenstra–Molnar–

e Weger exploited

rogue CA cert.

ame: new MD5 attack.

y 1996, a few years

e introduction of MD5,

and Dobbertin were

or MD5 to be scrapped.

ys attention to

alysis and makes

servative choices

ographic 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

until 201

Example

1024: “t

risk of k

performa

Example

to use se

Example

<https://>

is protec

<https://>

turns off

<http://>

crypto failures

rov-

ra-Molnar-

exploited

a cert.

MD5 attack.

few years

tion of MD5,

ertin were

to be scrapped.

on to

makes

choices

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 us

until 2013 switch

Example 3: DNSS

1024: "tradeoff be

risk of key compro

performance..."

Example 4: OpenS

to use secret AES

Example 5:

<https://sourcefo>

is protected by SS

<https://sourcefo>

turns off crypto: r

<http://sourcefor>

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 continued to use secret AES load address

Example 5:

<https://sourceforge.net/a> is protected by SSL but <https://sourceforge.net/d> turns off crypto: redirects to <http://sourceforge.net/de>

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 continues to use 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>.

performance problems
ad users to reduce
graphic security levels
up on cryptography.

Example 1: Google SSL
RSA-1024 until 2013.

note:
in 2003 concluded
RSA-1024 was breakable;
2003 Shamir–Tromer
cost 1 year, $\approx 10^7$ USD.
DHS and NIST response:
switch 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 continues
to use 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 ha
e.g. cry
enc
e.g. no
not

ce problems
to reduce
urity levels
tography.

e SSL

ntil 2013.

concluded
is breakable;
-Tromer
 $\approx 10^7$ USD.
ST response:
8 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 continues
to use 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-s
e.g. `crypto_box`
encrypts *and*
e.g. no RSA-1024
not even RSA

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 continues to use 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 op

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 continues to use 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 continues to use 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 \Rightarrow

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 continues to use 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 \Rightarrow 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.

e 2: Tor used RSA-1024
13 switch to Curve25519.

e 3: DNSSEC uses RSA-
tradeoff between the
key compromise and
performance...”

e 4: OpenSSL continues
secret AES load addresses.

e 5:
[/sourceforge.net/account](https://sourceforge.net/account)
ected by SSL but
[/sourceforge.net/develop](https://sourceforge.net/develop)
F crypto: redirects to
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 \Rightarrow
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 op
for any o
using AM
CPU (\$1
crypto_
crypto_
crypto_
crypto_

used RSA-1024
to Curve25519.

EC uses RSA-
between the
promise and

SSL continues
load addresses.

chrome.net/account

L but

chrome.net/develop

redirects to

chrome.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 \Rightarrow
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 p
for any common p
using AMD Pheno
CPU (\$190 in 201

`crypto_box`: >80

`crypto_box_oper`

`crypto_sign_ope`

`crypto_sign`: >1

024
5519.
RSA-
e

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 \Rightarrow
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 1
CPU (\$190 in 2011):

`crypto_box`: >80000.
`crypto_box_open`: >80000
`crypto_sign_open`: >7000
`crypto_sign`: >180000.

ues
esses.

ccount
velop
velop.

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 \Rightarrow
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 \Rightarrow
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.

no low-security options.

`crypto_box` always

encrypts *and* authenticates.

RSA-1024;

even RSA-2048.

ing risk:

and NaCl too slow \Rightarrow

no low-security libraries

rely on `crypto` entirely.

NaCl avoids this risk:

is exceptionally fast.

is faster than other libraries.

is *compatible 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

1. Pure

for any p

80000 15

fill up a

2. Pure

for many

from sam

if applica

`crypto_`

`crypto_`

`crypto_`

security options.

always

authenticates.

;

A-2048.

too slow \Rightarrow

security libraries

entirely.

this risk:

really fast.

other libraries.

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

1. Pure secret-key
for any packet size

80000 1500-byte p
fill up a 1 Gbps lin

2. Pure secret-key
for many packets

from same public
if application splits

crypto_box into
crypto_box_befo

crypto_box_aft

tions.

ates.

es

ries.

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 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`.

operations per second
common packet size,
AMD Phenom II X6 1100T
(90 in 2011):

crypto_box: >80000.

crypto_box_open: >80000.

crypto_sign_open: >70000.

crypto_sign: >180000.

arbitrary packet floods
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
of forged packets
under known key
no time to

(This does not work
for forged packets
but flooding
continues to
to *know*

4. Fast
doubling
crypto_box
for valid

per second
packet size,
om II X6 1100T
1):
0000.
n: >80000.
en: >70000.
180000.
packet floods
per CPU,
ocol 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 reject
of forged packets
under known public
no time spent on c
(This doesn't help
for forgeries under
but flooded server
continue providing
to *known* keys.)
4. Fast batch veri
doubling speed of
crypto_sign_ope
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: 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.

t, it's even faster!

secret-key crypto

packet size:

500-byte packets/second

1 Gbps link.

secret-key crypto

y packets

me public key,

ation splits

_box into

_box_beforenm and

_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.

Cryptog

The mai
achieve
without

ECC, no
much st
Curve25
curves:

Salsa20,
much lar
Poly1305
informat
EdDSA,
collision-

faster!
crypto
e:
packets/second
nk.
crypto
key,
s
orenm and
ernm.

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.

Cryptographic det
The main NaCl wo
achieve very high s
without compromi
ECC, not RSA:
much stronger sec
Curve25519, not M
curves: [safecurv](#)
Salsa20, not AES:
much larger securi
Poly1305, not HM
information-theore
EdDSA, not ECDS
collision-resilience

cond

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.

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.yt

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.

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.jp.to

Salsa20, not AES:

much larger security margin.

Poly1305, not HMAC:

information-theoretic security.

EdDSA, not ECDSA:

collision-resilience et al.

fast rejection
d packets
known public keys:
spent on decryption.
doesn't help much
eries under *new* keys,
ded server can
e providing fast service
n keys.)
batch verification,
; speed of
_sign_open
signatures.

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.jp.to

Salsa20, not AES:
much larger security margin.

Poly1305, not HMAC:
information-theoretic security.

EdDSA, not ECDSA:
collision-resilience et al.

Speed e

Oops, th
users co
of how f

tion

ic keys:
decryption.

much
new keys,
can
fast service

fication,

en

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.jp.to

Salsa20, not AES:
much larger security margin.

Poly1305, not HMAC:
information-theoretic security.

EdDSA, not ECDSA:
collision-resilience et al.

Speed education

Oops, there's another
users completely unaware
of how **fast** crypto

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.jp.to

Salsa20, not AES:

much larger security margin.

Poly1305, not HMAC:

information-theoretic security.

EdDSA, not ECDSA:

collision-resilience et al.

Speed education

Oops, there's another risk:
users completely unaware
of how **fast** crypto can be.

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.

Speed education

Oops, there's another risk:
users completely unaware
of how **fast** crypto can be.

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.

Speed education

Oops, there's another risk:
users completely unaware
of how **fast** crypto can be.

Example:

“PRESERVE contributes to the security and privacy of future vehicle-to-vehicle and vehicle-to-infrastructure communication systems by addressing critical issues like performance, scalability, and deployability of V2X security systems.”

preserve-project.eu

graphic details

n NaCl work we did:
very high speeds
compromising security.

t RSA:

stronger security record.

519, not NSA/NIST

safecurves.cr.yp.to

not AES:

arger security margin.

5, not HMAC:

tion-theoretic security.

not ECDSA:

-resilience et al.

Speed education

Oops, there's another risk:
users completely unaware
of how **fast** crypto can be.

Example:

“PRESERVE contributes to the security and privacy of future vehicle-to-vehicle and vehicle-to-infrastructure communication systems by addressing critical issues like performance, scalability, and deployability of V2X security systems.”

preserve-project.eu

*“[In] mo
the pack
750 pack
maximum
goes we
(2,265 p
Processi
second a
ms can
hardware
a Pentiu
needs ab
a verific
cryptogr
likely to*

ails

ork we did:
speeds
ising security.

urity record.
NSA/NIST
[es.cr.yp.to](https://www.cr.yp.to)

ty margin.
IAC:
etic security.
SA:
et al.

Speed education

Oops, there's another risk:
users completely unaware
of how **fast** crypto can be.

Example:

"PRESERVE contributes to the security and privacy of future vehicle-to-vehicle and vehicle-to-infrastructure communication systems by addressing critical issues like performance, scalability, and deployability of V2X security systems."

preserve-project.eu

"[In] most driving the packet rates do 750 packets per second maximum highway goes well beyond 2,265 packets per second. Processing 1,000 packets per second and processing times can hardly be met by hardware. As discussed, a Pentium D 3.4 GHz needs about 5 times a verification ... a cryptographic co-processor likely to be necessary."

Speed education

Oops, there's another risk: users completely unaware of how **fast** crypto can be.

Example:

“PRESERVE contributes to the security and privacy of future vehicle-to-vehicle and vehicle-to-infrastructure communication systems by addressing critical issues like performance, scalability, and deployability of V2X security systems.”

preserve-project.eu

“[In] most driving situations the packet rates do not exceed 750 packets per second. On maximum highway scenario goes well beyond this value (2,265 packets per second).

Processing 1,000 packets per second and processing each ms can hardly be met by current hardware. As discussed in [3] a Pentium D 3.4 GHz processor needs about 5 times as long a verification ... a dedicated cryptographic co-processor is likely to be necessary.”

Speed education

Oops, there's another risk: users completely unaware of how **fast** crypto can be.

Example:

“PRESERVE contributes to the security and privacy of future vehicle-to-vehicle and vehicle-to-infrastructure communication systems by addressing critical issues like performance, scalability, and deployability of V2X security systems.”

preserve-project.eu

“[In] most driving situations . . . the packet rates do not exceed 750 packets per second. Only the maximum highway scenario . . . goes well beyond this value (2,265 packets per second). . . .

Processing 1,000 packets per second and processing each in 1 ms can hardly be met by current hardware. As discussed in [32], a Pentium D 3.4 GHz processor needs about 5 times as long for a verification . . . a dedicated cryptographic co-processor is likely to be necessary.”

Other risk:
unaware
can be.

Contributes to the
efficiency of future
and vehicle-
communication
requiring critical
performance, scalability,
of V2X security

ct.eu

*“[In] most driving situations . . .
the packet rates do not exceed
750 packets per second. Only the
maximum highway scenario . . .
goes well beyond this value
(2,265 packets per second). . . .*

*Processing 1,000 packets per
second and processing each in 1
ms can hardly be met by current
hardware. As discussed in [32],
a Pentium D 3.4 GHz processor
needs about 5 times as long for
a verification . . . a dedicated
cryptographic co-processor is
likely to be necessary.”*

“NEON crypto” (Cortex-A9)
on 1GHz Cortex A9
5.48 cycles/byte (SHA-1)
2.30 cycles/byte (SHA-256)
for Salsa20, Poly1305
498349 cycles (2000000000/4090)
624846 cycles (1600000000/2560)
for Curve25519 DHE

“[In] most driving situations . . . the packet rates do not exceed 750 packets per second. Only the maximum highway scenario . . . goes well beyond this value (2,265 packets per second). . . .

Processing 1,000 packets per second and processing each in 1 ms can hardly be met by current hardware. As discussed in [32], a Pentium D 3.4 GHz processor needs about 5 times as long for a verification . . . a dedicated cryptographic co-processor is likely to be necessary.”

“NEON crypto” (CHES 2011) on 1GHz Cortex A8 core:
5.48 cycles/byte (1.4 Gbps)
2.30 cycles/byte (3.4 Gbps)
for Salsa20, Poly1305.
498349 cycles (2000/second)
624846 cycles (1600/second)
for Curve25519 DH, verify.

“[In] most driving situations . . . the packet rates do not exceed 750 packets per second. Only the maximum highway scenario . . . goes well beyond this value (2,265 packets per second). . . .

Processing 1,000 packets per second and processing each in 1 ms can hardly be met by current hardware. As discussed in [32], a Pentium D 3.4 GHz processor needs about 5 times as long for a verification . . . a dedicated cryptographic co-processor is likely to be necessary.”

“NEON crypto” (CHES 2012)
on 1GHz Cortex A8 core:
5.48 cycles/byte (1.4 Gbps),
2.30 cycles/byte (3.4 Gbps)
for Salsa20, Poly1305.
498349 cycles (2000/second),
624846 cycles (1600/second)
for Curve25519 DH, verify.

“[In] most driving situations . . . the packet rates do not exceed 750 packets per second. Only the maximum highway scenario . . . goes well beyond this value (2,265 packets per second). . . .

Processing 1,000 packets per second and processing each in 1 ms can hardly be met by current hardware. As discussed in [32], a Pentium D 3.4 GHz processor needs about 5 times as long for a verification . . . a dedicated cryptographic co-processor is likely to be necessary.”

“NEON crypto” (CHES 2012)
on 1GHz Cortex A8 core:
5.48 cycles/byte (1.4 Gbps),
2.30 cycles/byte (3.4 Gbps)
for Salsa20, Poly1305.
498349 cycles (2000/second),
624846 cycles (1600/second)
for Curve25519 DH, verify.
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).

“[In] most driving situations . . . the packet rates do not exceed 750 packets per second. Only the maximum highway scenario . . . goes well beyond this value (2,265 packets per second). . . .

Processing 1,000 packets per second and processing each in 1 ms can hardly be met by current hardware. As discussed in [32], a Pentium D 3.4 GHz processor needs about 5 times as long for a verification . . . a dedicated cryptographic co-processor is likely to be necessary.”

“NEON crypto” (CHES 2012)
on 1GHz Cortex A8 core:
5.48 cycles/byte (1.4 Gbps),
2.30 cycles/byte (3.4 Gbps)
for Salsa20, Poly1305.
498349 cycles (2000/second),
624846 cycles (1600/second)
for Curve25519 DH, verify.
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.

*most driving situations ...
packet rates do not exceed
packets per second. Only the
m highway scenario ...
All beyond this value
packets per second). ...
ing 1,000 packets per
and processing each in 1
hardly be met by current
e. As discussed in [32],
m D 3.4 GHz processor
out 5 times as long for
ation ... a dedicated
raphic co-processor is
be necessary.”*

“NEON crypto” (CHES 2012)
on 1GHz Cortex A8 core:
5.48 cycles/byte (1.4 Gbps),
2.30 cycles/byte (3.4 Gbps)
for Salsa20, Poly1305.
498349 cycles (2000/second),
624846 cycles (1600/second)
for Curve25519 DH, verify.
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.

Case stu
1985 EIC
(R, S) is
if $B^{H(M)}$
and R, S
Here q is
 B is star
 A is sign
 $H(M)$ is
Signer g
as secret
easily so

*situations . . .
do not exceed
second. Only the
scenario . . .
this value
r second). . . .
packets per
sing each in 1
met by current
ussed in [32],
GHz processor
es as long for
a dedicated
processor is
ary.”*

“NEON crypto” (CHES 2012)
on 1GHz Cortex A8 core:
5.48 cycles/byte (1.4 Gbps),
2.30 cycles/byte (3.4 Gbps)
for Salsa20, Poly1305.
498349 cycles (2000/second),
624846 cycles (1600/second)
for Curve25519 DH, verify.
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.

Case study: EdDSA
1985 ElGamal signature
 (R, S) is signature
if $B^{H(M)} \equiv A^R R^S$
and $R, S \in \{0, 1, \dots\}$
Here q is standard
 B is standard base
 A is signer's public
 $H(M)$ is hash of message
Signer generates A
as secret powers of
easily solves for S .

“NEON crypto” (CHES 2012)

on 1GHz Cortex A8 core:

5.48 cycles/byte (1.4 Gbps),

2.30 cycles/byte (3.4 Gbps)

for Salsa20, Poly1305.

498349 cycles (2000/second),

624846 cycles (1600/second)

for Curve25519 DH, verify.

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.

Case study: EdDSA

1985 ElGamal signatures:

(R, S) is signature of M

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

and $R, S \in \{0, 1, \dots, q-2\}$

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 .

“NEON crypto” (CHES 2012)

on 1GHz Cortex A8 core:

5.48 cycles/byte (1.4 Gbps),

2.30 cycles/byte (3.4 Gbps)

for Salsa20, Poly1305.

498349 cycles (2000/second),

624846 cycles (1600/second)

for Curve25519 DH, verify.

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.

Case study: EdDSA

1985 ElGamal signatures:

(R, S) is signature of M

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

and $R, S \in \{0, 1, \dots, q - 2\}$.

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 .

crypto" (CHES 2012)

Cortex A8 core:

1.4 Gbps),

3.4 Gbps)

20, Poly1305.

cycles (2000/second),

cycles (1600/second)

e25519 DH, verify.

Cortex A8 was high-end

one core in 2010: e.g.,

Exynos 3110 (Galaxy S);

AP3630 (Motorola Droid

le A4 (iPad 1/iPhone 4).

llwinner A13, \$5 in bulk.

Case study: EdDSA

1985 ElGamal signatures:

(R, S) is signature of M

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

and $R, S \in \{0, 1, \dots, q - 2\}$.

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 Sci

1. Hash

$B^{H(M)} \equiv$

Reduces

2. Repla

with two

$B^{H(M)}/$

Saves tim

3. Simp

$B^{H(M)}/$

Saves tim

4. Merg

$B^{H(R,M)}$

\Rightarrow Resili

CHES 2012)

8 core:

1.4 Gbps),

3.4 Gbps)

305.

00/second),

00/second)

H, verify.

was high-end

n 2010: e.g.,

3110 (Galaxy S);

Motorola Droid

d 1/iPhone 4).

13, \$5 in bulk.

Case study: EdDSA

1985 ElGamal signatures:

(R, S) is signature of M

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

and $R, S \in \{0, 1, \dots, q - 2\}$.

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 impr

1. Hash R in the

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

Reduces attacker c

2. Replace three e

with two exponent

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

Saves time in verif

3. Simplify by rela

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

Saves time in verif

4. Merge the hash

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

\Rightarrow Resilient to H

Case study: EdDSA

1985 ElGamal signatures:

(R, S) is signature of M

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

and $R, S \in \{0, 1, \dots, q - 2\}$.

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 $R, S \in \{0, 1, \dots, q - 2\}$.

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.

EdDSA

Gamal signatures:

Signature of M

$$S \equiv A^R R^S \pmod{q}$$

$$S \in \{0, 1, \dots, q - 2\}.$$

Standard prime,

Standard base,

Sender's public key,

Hash of message.

Generates A and R

Random powers of B ;

Looks 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:

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

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

$$B^S \equiv R^S$$

Simpler,

6. Compact

Saves space

7. Use hash

Saves space

A

atures:

e of M

(mod q)

$\dots, q - 2$ }.
prime,

e,

c key,

message.

A and R

f B ;

.

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 inverse:

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

Simpler, faster.

6. Compress R to

Saves space in signature.

7. Use half-size H

Saves space in signature.

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 s

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

Simpler, faster.

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

Saves space in signatures.

7. Use half-size H output.

Saves space in signatures.

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:

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

Simpler, faster.

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

Saves space in signatures.

7. Use half-size H output.

Saves space in signatures.

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:

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

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:

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

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:

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

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.

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:

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

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.

Patent expired in 2008.

Schnorr improvements:

R in the exponent:

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

attacker control.

Use three exponents

two exponents:

$$H(R) \equiv AR^S/H(R).$$

time in verification.

Simplify by relabeling S :

$$H(R) \equiv AR^S.$$

time in verification.

Use the hashes:

$$) \equiv AR^S.$$

resistant 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.

Patent expired in 2008.

EdDSA

Duif-Lam

Use elliptic

–1-twist

\Rightarrow very

natural s

no excep

Skip sign

Support

Use dou

and inclu

Generate

as a secr

\Rightarrow Avoid

Improvements:

exponent:

S .

control.

exponents

es:

$RS/H(R)$.

ification.

labeling S :

RS .

ification.

nes:

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.

Patent expired in 2008.

EdDSA (CHES 2002)

Duif-Lange-Schwa

Use elliptic curves

-1-twisted Edwards

\Rightarrow very high speed

natural side-chann

no exceptional cas

Skip signature con

Support batch ver

Use double-size H

and include A as i

Generate R determ

as a secret hash o

\Rightarrow Avoid PlayStat

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.

Patent expired in 2008.

EdDSA (CHES 2011 Bernstein, Duif–Lange–Schwabe–Yang)

Use elliptic curves in “compact–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.

Patent expired in 2008.

EdDSA (CHES 2011 Bernstein–Duif–Lange–Schwabe–Yang):

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.