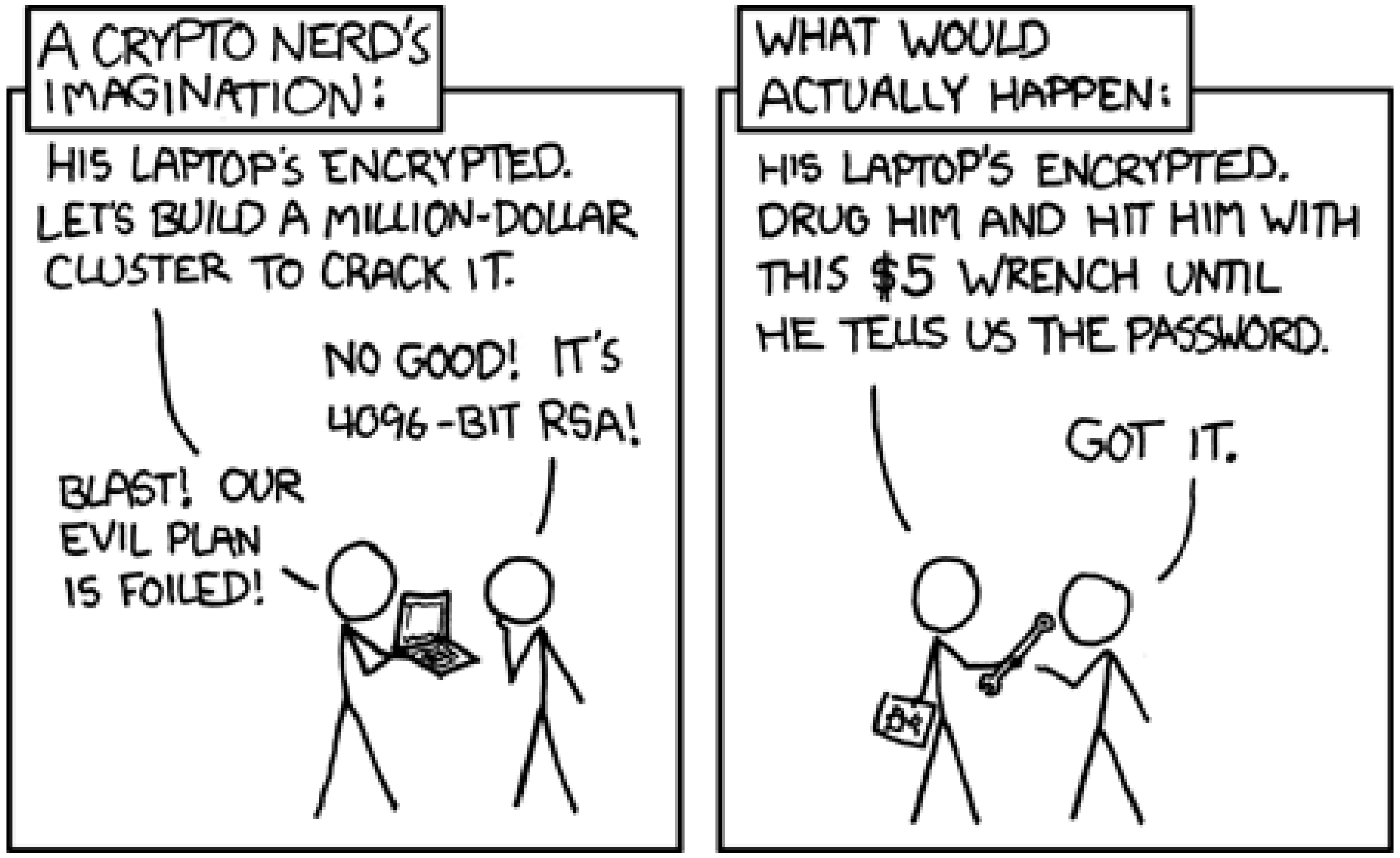
The security impact of a new cryptographic library

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.

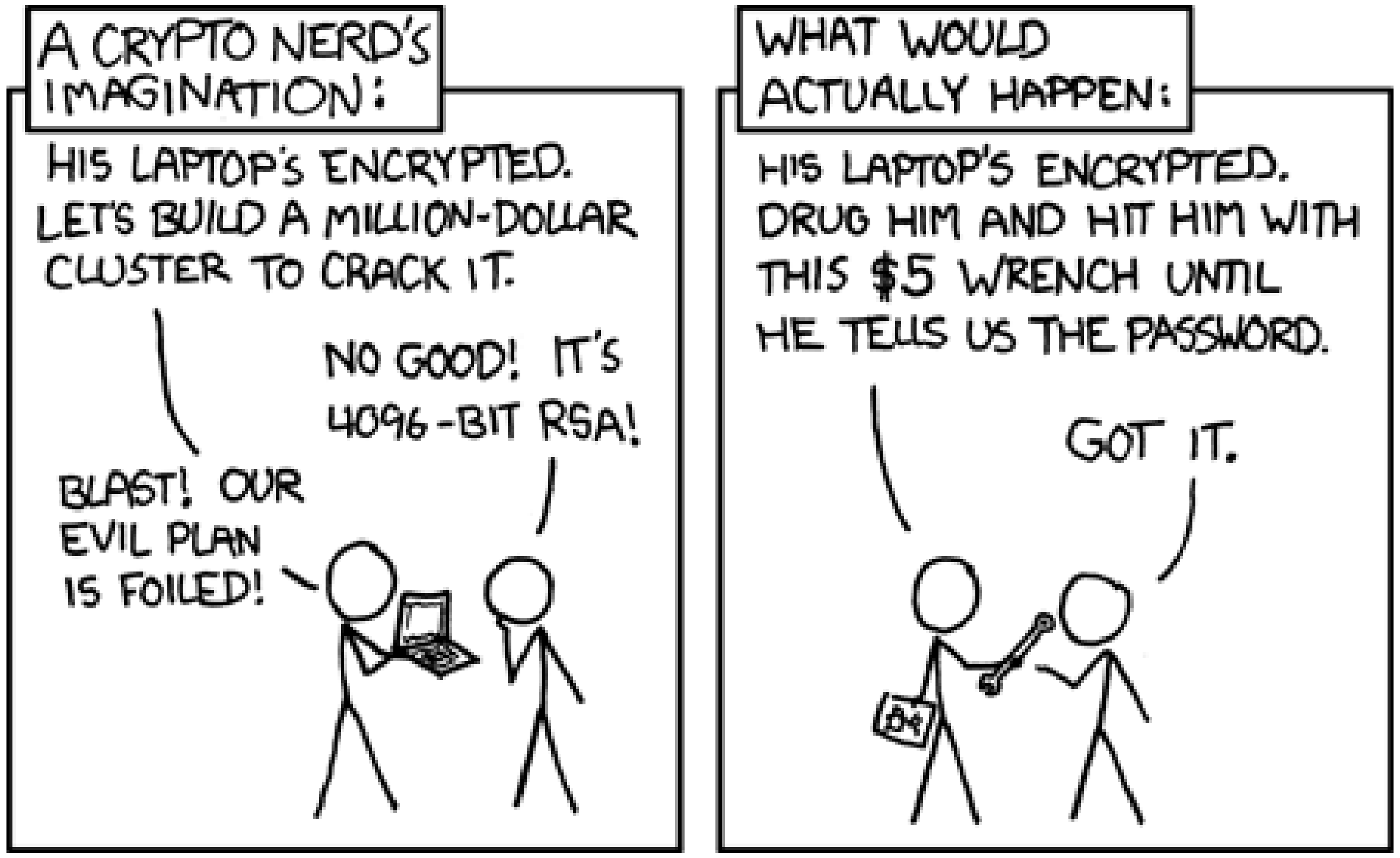
Implementations are available in public cryptographic libraries such as OpenSSL.

Common security practice is to use those implementations.

The security impact of a new cryptographic library

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.

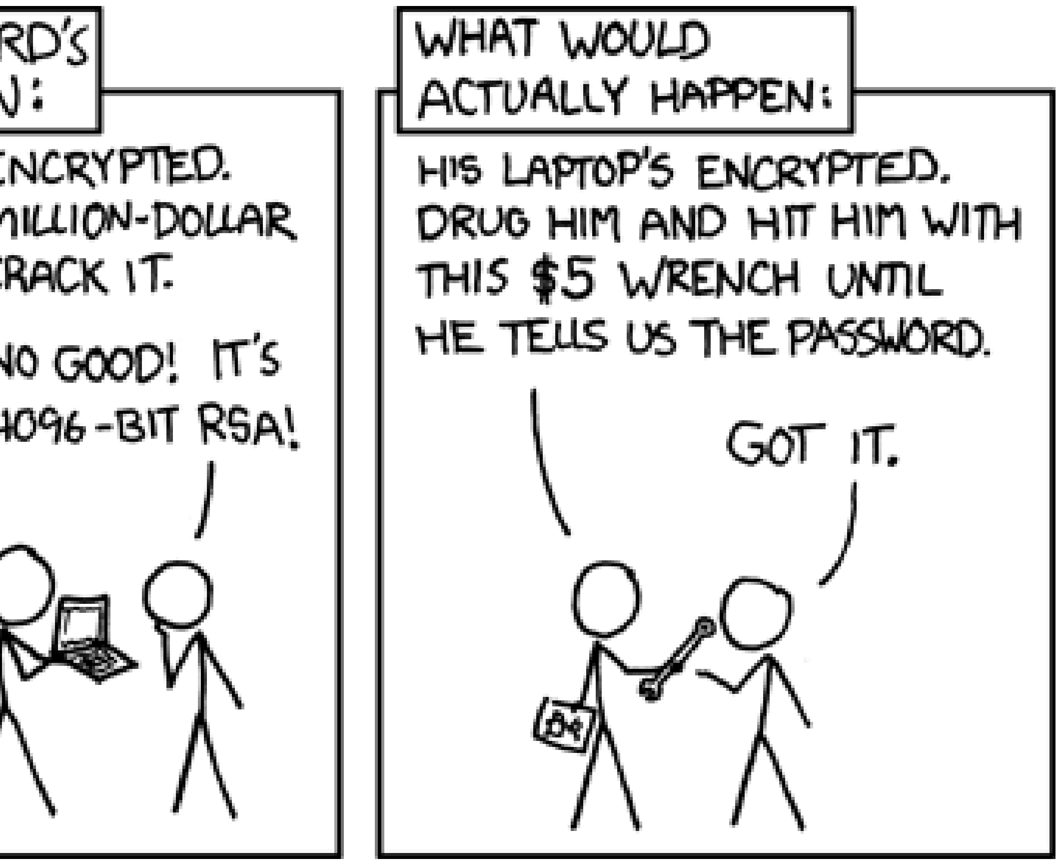
urity impact
cryptographic library

ernstein, U. Illinois Chicago
Eindhoven

ange, T. U. Eindhoven

rk with:

hwabe, R. U. Nijmegen



/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.cr
and exte

Acknowl
code cor
Matthew
Media),
Emilia K
Adam La
Bo-Yin

ct

aphic library

. Illinois Chicago

n

J. Eindhoven

. U. Nijmegen

n/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 designed
a new cryptograph
NaCl (“salt”), to
the underlying pro

nacl.cr.yp.to:
and extensive docu

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

ry

hicago

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.cr.yp.to`: source
and extensive documentatio

Acknowledgments:
code contributions from
Matthew Dempsky (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.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.

ntations are available
c cryptographic libraries
OpenSSL.

n security practice is
hose implementations.

ptography is still
er! Complete failures
dentiality and integrity.

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

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

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

Most of
is crypto
Primary

Main tas
authent

Alice ha

Uses Bo
Alice's s
authenti
Sends *c*

Bob use
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.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 cryptographicall
Primary goal of N

Main task: **public**
authenticated en

Alice has a messag

Uses Bob's public
Alice's secret key
authenticated ciph
Sends c to Bob.

Bob uses Alice's p
and Bob's secret k
to verify and recov

ds.

k

le
ries

.

ns.

s
ty.

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

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

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

Most of the Internet
is cryptographically unprotec
Primary goal of NaCl: Fix th

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

Alice has a message *m* for B

Uses Bob's public key and
Alice's secret key to comput
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.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
ryptographic library,
salt”), to address
erlying problems.

`r.yp.to`: source
ensive documentation.

edgments:
ntributions from
v Dempsky (Mochi
Niels Duif (Eindhoven),
Käsper (Leuven),
angley (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 usi
typical c

Generate
Use AES
Hash en
Read RS
Use key
Read Bo
Use key
Convert

Plus mo
allocate
handle e

+implemented
nic library,
address
blems.

source
umentation.

:
s from
y (Mochi
f (Eindhoven),
uven),
oogle),
demia 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 e
Hash encrypted pa
Read RSA key fro
Use key to sign ha
Read Bob's key fr
Use key to encryp
Convert to wire fo

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

nted

n.

en),

ica).

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 pac
Hash encrypted packet.
Read RSA key from wire for
Use key to sign hash.
Read Bob's key from wire fo
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
ographically unprotected.
goal of NaCl: Fix this.

sk: **public-key**
ticated encryption.

s a message *m* for Bob.

b's public key and
ecret key to compute
cated ciphertext *c*.
to Bob.

s Alice's public key
's secret key
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 usi

`c = cry`

net
y unprotected.
aCl: Fix this.

-key
ncryption.

ge m for Bob.

key and
to compute
nertext c.

ublic key
key
ver 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(`

cted.

his.

Bob.

e

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

ryptographic library:

e random AES key.

S key to encrypt packet.

crypted 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,

rrors, 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
```

hic 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_`

:

ket.

mat.

rmat.

e etc.

Alice using NaCl:

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

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

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

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

Bob verifying, decrypting:

`m=crypto_box_open(c,n,`

Initial key generation:

`pk = 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:

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

secret key `sk`.

public key `pk`.

nonce `n`.

ytes longer than `m`.

cts are C++

`ring` 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)

on.

ers;

ilures.

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 mor

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.

fying, decrypting:

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

ey generation:

`ypto_box_keypair(&sk)`

ead use **signatures**
c messages:

`ypto_sign_keypair(&sk)`
secret key,
public key.

`ypto_sign(m,sk)`
overhead.

`pto_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.

<u>No secre</u>

2005 Os
65ms to
used for
Attack p
but with

Almost a
use fast
Kernel’s
influence
influenci
influenci
of the at
65ms to

rypting:

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

ion:

`_keypair(&sk)`

gnatures

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–Shami
65ms to steal Linu
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 measu
of the attack proc
65ms to compute

```
pk,sk)

(&sk)

r(&sk)

,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 addresses

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

Almost all AES implementat
use fast lookup tables.
Kernel’s secret AES key
influences table-load addres
influencing CPU cache state
influencing measurable timi
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}.

ounds too simple!
pplications need more?”

es of applications
aCl’s `crypto_box`:

rve 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!

s need more?”

cations

`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 cryptographi

still use secret loa

but add “counterm

intended to obscu

upon the CPU ca

Not confidence-ins

likely to be breaka

re?”

ement.

p.

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 addresse
but add “countermeasures”
intended to obscure influenc
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.

et load addresses

vik–Shamir–Tromer:
steal Linux AES key
hard-disk encryption.
process on same CPU
hout privileges.

all AES implementations
lookup tables.
secret AES key
es table-load addresses,
ng CPU cache state,
ng measurable timings
ttack 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 secre

2011 Br
minutes
machine
Secret b
influence

Most cry
has man
variation
e.g., `mem`

dresses

ir–Tromer:
ux AES key
encryption.
same CPU
eges.

nplementations
bles.
ES 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 cryptographi
has many more sm
variations in timin
e.g., `memcmp` for I

t

y

n.

U

tions

ses,

e,

ngs

$^{-1}$.

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 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.

yptographic libraries
secret load addresses
“countermeasures”
l to obscure influence
e CPU cache state.
fidence-inspiring;
be breakable.

stematically avoids
s from addresses
pend 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 ≈10^6

SSL first
then che
(which r
Subsequ
more ser

Server re
pattern
pattern

ic libraries
d addresses
neasures”
re influence
che state.
spiring;
ble.

ly avoids
resses
cret data.
pe 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 serve
to $\approx 10^6$ variants c

SSL first inverts R
then checks for “P
(which many forge
Subsequent proces
more serious integ

Server responses r
pattern of PKCS f
pattern reveals pla

es

e

ter.

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 ≈10^6 variants of cipherte

SSL first inverts RSA,
then checks for "PKCS padd
(which many forgeries have)
Subsequent processing appli
more serious integrity check

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.

et branch conditions

umley–Tuveri:
to steal another
’s OpenSSL ECDSA key.
ranch conditions
e timings.

yptographic software
y more small-scale
ns in timing:
`mcmp` for IPsec MACs.

stematically avoids
ch conditions
pend on secret data.
es 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
subseque

But hard
BEAST,

conditions

veri:
nother
SL ECDSA key.
ditions

ic software
nall-scale
g:
Psec MACs.

ly avoids
ons
cret data.
pe 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 c
subsequent integri

But hard to get th
BEAST, Lucky 13

key.

e

s.

ter.

No padding oracles

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

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

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

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

But hard to get this right: s
BEAST, Lucky 13, POODL

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
BEAST, Lucky 13, POODLE, etc.

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
BEAST, Lucky 13, POODLE, etc.

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.

ling oracles

eichenbacher:
SSL RSA ciphertext
ving server responses
variants of ciphertext.

t inverts RSA,
ecks for “PKCS padding”
many forgeries have).
ent processing applies
rious integrity checks.

esponses 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
BEAST, Lucky 13, POODLE, etc.

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
randomn

s

her:
ciphertext
r responses
of ciphertext.

SA,
PKCS padding”
eries have).
ssing applies
rity checks.

eveal
forgeries;
intext.

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

But hard to get this right: see BEAST, Lucky 13, POODLE, etc.

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 rando

2008 Bello: Debia
OpenSSL keys for
had only 15 bits o

Debian developer
a subtle line of Op
randomness-genera

t
es
xt.

ding"
.
es
s.

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

But hard to get this right: see
BEAST, Lucky 13, POODLE, etc.

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 remov
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
BEAST, Lucky 13, POODLE, etc.

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
BEAST, Lucky 13, POODLE, etc.

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:
de differences
padding checks and
ent integrity checks.

d to get this right: see
Lucky 13, POODLE, etc.

es not decrypt
nessage is authenticated.
tion procedure rejects
ries in constant time.
are further constrained
once key separation
ndard 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.

Centraliz

merge m
pool fee

Merging
auditabl
bad/faili
if there i

rategy:
nces
checks and
ty checks.

his right: see
, POODLE, etc.

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 entro
pool feeding many

Merging is determ
auditable. Can sur
bad/failing/malicio
if there is one goo

d

see
E, etc.

nted.
s
e.
ned

g.

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 source
pool feeding many applicati

Merging is deterministic and
auditable. Can survive man
bad/failing/malicious source
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.

zing randomness

llo: Debian/Ubuntu
L keys for 1.5 years
y 15 bits of entropy.

developer had removed
line of OpenSSL
ness-generating code.

es `/dev/urandom`,
random-number generator.
ng this kernel code
more tractable than
g separate RNG code
security library.

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

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

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

Avoiding

2010 Bu
Sven: Sc
requirem
for each
leaked P

omness

n/Ubuntu
1.5 years
f entropy.

had removed
penSSL
ating code.

random,
umber generator.
nel code
table than
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 ne
for each signature
leaked PS3 code-s

ved

e.

erator.

n

e

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 rando

2010 Bushing–Marcan–Segh
Sven: Sony ignored ECDSA
requirement of new random
for each signature. ⇒ Signa
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. ⇒ 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. ⇒ 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).

zation allows OS to
nany entropy sources into
ding many applications.

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

ep backwards:
DRAND in applications.
ntropy 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. ⇒ 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–de
MD5 ⇒

ws OS to
py sources into
applications.

inistic and
rvive many
ous sources
d source.

rds:
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. ⇒ 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 ex
MD5 ⇒ rogue CA

s into
ons.

l

y

es

ions.
ackup;

NaCl.

Avoiding unnecessary randomness

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

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

Also simplifies testing. NaCl uses automated test battery from eBACS (ECRYPT Benchmarking of Cryptographic Systems).

Avoiding pure crypto failure

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

Avoiding unnecessary randomness

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

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

Also simplifies testing. NaCl uses automated test battery from eBACS (ECRYPT Benchmarking of Cryptographic Systems).

Avoiding pure crypto failures

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

Avoiding unnecessary randomness

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

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

Also simplifies testing. NaCl uses automated test battery from eBACS (ECRYPT Benchmarking of Cryptographic Systems).

Avoiding pure crypto failures

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

ushing–Marcan–Segher–
ony ignored ECDSA
nent of new randomness
signature. ⇒ Signatures
S3 code-signing key.

s *deterministic*
_box and `crypto_sign`.
ness only for `keypair`.
es this type of disaster.

nplifies testing. NaCl uses
zed test battery from
(ECRYPT Benchmarking
ographic 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 p
often lea
cryptogr
or give u

Example
used RS

Security
Analyses
that RS/
e.g., 200
estimate
RSA Lab
Move to

ary randomness

rcan–Segher–
d ECDSA
w randomness
. ⇒ Signatures
igning key.

istic
`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 performanc
often lead users to
cryptographic secu
or give up on cryp

Example 1: Googl
used RSA-1024 ur

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

mness

er–

ness
tures
/.

`ign`.
`ir`.
ter.

l uses
n
rking

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 problem
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 breakab
e.g., 2003 Shamir–Tromer
estimated 1 year, $\approx 10^7$ USD
RSA Labs and NIST respons
Move to RSA-2048 by 2010.

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 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, ≈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://s

to failures

rov–
ra–Molnar–
xploited
cert.
MD5 attack.

few years
tion of MD5,
ertin were
be scrapped.

n to
makes
choices
rimitives.

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: Open
to use secret AES

Example 5:
https://sourcefo
is protected by SS
https://sourcefo
turns off crypto:
http://sourcefor

s

–

ck.

5,

ped.

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-1
until 2013 switch to Curve2

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

Example 4: OpenSSL contin
to use secret AES load addr

Example 5:
https://sourceforge.net/a
is protected by SSL but
https://sourceforge.net/d
turns off crypto: redirects t
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
raphic security levels
up on cryptography.

e 1: Google SSL
A-1024 until 2013.

note:
s in 2003 concluded
A-1024 was breakable;
03 Shamir–Tromer
ed 1 year, $\approx 10^7$ USD.
ps and NIST response:
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
reduce
urity levels
tography.

e SSL
ntil 2013.

concluded
ns breakable;
-Tromer
≈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

ns

le;

).

se:

.

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* authentic

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 ⇒ 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 ⇒ 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
ey compromise and
ance. . . ”

e 4: OpenSSL continues
ecret AES load addresses.

e 5:
`/sourceforge.net/account`
cted by SSL but
`/sourceforge.net/develop`
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 ⇒
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
using A
CPU ($1

`crypto_`

`crypto_`

`crypto_`

`crypto_`

sed RSA-1024
to Curve25519.

EC uses RSA-
etween the
mise and

SSL continues
load addresses.

`rge.net/account`
L but
`rge.net/develop`
edirects to
`ge.net/develop`.

NaCl has no low-security options.

e.g. `crypto_box` always encrypts *and* authenticates.

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

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

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

NaCl operations p
for any common p
using AMD Phen
CPU ($190 in 201

`crypto_box`: >8

`crypto_box_ope`

`crypto_sign_ope`

`crypto_sign`: $>$

024
5519.

RSA-
e

nues
esses.

ccount

evelop
o
evelop.

NaCl has no low-security options.

e.g. `crypto_box` always
encrypts *and* authenticates.

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

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

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

NaCl operations per second
for any common packet size
using AMD Phenom II X6 1
CPU ($190 in 2011):

`crypto_box`: >80000.

`crypto_box_open`: >8000

`crypto_sign_open`: >700

`crypto_sign`: >180000.

NaCl has no low-security options.

e.g. `crypto_box` always
encrypts *and* authenticates.

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

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

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

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

`crypto_box`: $>$80000.

`crypto_box_open`: $>$80000.

`crypto_sign_open`: $>$70000.

`crypto_sign`: $>$180000.

NaCl has no low-security options.

e.g. `crypto_box` always
encrypts *and* authenticates.

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

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

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

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

`crypto_box`: $>$80000.

`crypto_box_open`: $>$80000.

`crypto_sign_open`: $>$70000.

`crypto_sign`: $>$180000.

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

s no low-security options.
`rypto_box` always
rypts *and* authenticates.
RSA-1024;
even RSA-2048.

ng risk:
nd NaCl too slow ⇒
o low-security libraries
le crypto entirely.

Cl avoids this risk:
exceptionally fast.
ster than other libraries.
p 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 1
fill up a

2. Pure
for many
from sar
if applica
`crypto_`
`crypto_`
`crypto_`

ecurity options.
always
authenticates.
;
-2048.

o slow ⇒
rity libraries
ntirely.

this risk:
lly 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 li

2. Pure secret-key
for many packets
from same public
if application split
`crypto_box` into
`crypto_box_befo`
`crypto_box_afte`

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/se
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 $\approx$30 Mbps per CPU,
depending on protocol details.

But wait, it's even faster!

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

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

erations per second
common packet size,
MD Phenom II X6 1100T
190 in 2011):

`_box`: $>$80000.

`_box_open`: $>$80000.

`_sign_open`: $>$70000.

`_sign`: $>$180000.

arbitrary packet floods
30 Mbps per CPU,
ng 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
of forge
under k
no time

(This d
for forge
but floo
continue
to *know*

4. Fast
doubling
`crypto_`
for valid

er second
acket size,
om II X6 1100T
1):

0000.

n: >80000.

en: >70000.

180000.

packet floods
er 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 publi
no time spent on

(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

,
100T

0.

00.

ods

ls.

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 servi 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 la
Poly130
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 N
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 secur

ECC, not RSA:
much stronger security reco
Curve25519, not NSA/NIST
curves: safecurves.cr.yp
Salsa20, not AES:
much larger security margin
Poly1305, not HMAC:
information-theoretic securit
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.yp.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
nown public keys:
spent on decryption.

esn't help much
eries under *new* keys,
ded server can
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.yp.to`
Salsa20, not AES:
much larger security margin.
Poly1305, not HMAC:
information-theoretic security.
EdDSA, not ECDSA:
collision-resilience et al.

Speed ec

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.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 anot
users completely u
of how fast crypto

.

’,

ce

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.

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`

raphic details

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

t RSA:
ronger security record.
519, not NSA/NIST
safecurves.cr.yp.to
 not AES:
rger 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 pac
maximu
goes we
(2,265 p

Processi
second a
ms can
hardwar
a Pentiu
needs ab
a verifica
cryptogr
likely to

ails

ork we did:
speeds
ising security.

urity record.
NSA/NIST
`es.cr.yp.to`

ty margin.
AC:
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 d
750 packets per se
maximum highway
goes well beyond t
(2,265 packets pe

Processing 1,000
second and proces
ms can hardly be
hardware. As disc
a Pentium D 3.4
needs about 5 tim
a verification . . .
cryptographic co-
likely to be necess

:

rity.

rd.

-

.to

.

y.

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 exce
750 packets per second. On
maximum highway scenario
goes well beyond this value
(2,265 packets per second).*

*Processing 1,000 packets pe
second and processing each
ms can hardly be met by cu
hardware. As discussed in [3
a Pentium D 3.4 GHz proce
needs about 5 times as long
a verification . . . a dedicate
cryptographic co-processor i
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."

ducation

here's another risk:
mpletely unaware
ast crypto can be.

e:
ERVE contributes to the
and privacy of future
o-vehicle and vehicle-
structure communication
by addressing critical
ke performance, scalability,
loyability of V2X security
."

ve-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."

"NEON
on 1GH
5.48 cyc
2.30 cyc
for Sals
498349
624846
for Curv

ther risk:
naware
can be.

ributes to the
cy of future
and vehicle-
ommunication
sing critical
ance, 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" (
on 1GHz Cortex-A
5.48 cycles/byte (
2.30 cycles/byte (
for Salsa20, Poly1
498349 cycles (20
624846 cycles (16
for Curve25519 D

the
re
e-
ntion
al
ability,
curity

“*[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 201
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.

ost driving situations . . .
ket rates do not exceed
kets per second. Only the
m highway scenario . . .
ll 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],
um D 3.4 GHz processor
bout 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.

<u>Case stu</u>

1985 El
(R, S) is
if $B^{H(M}$
and R, S

Here q i
B is star
A is sign
$H(M)$ is

Signer g
as secret
easily so

situations . . .
lo not exceed
econd. Only the
y scenario . . .
this value
r second). . . .

packets per
ssing each in 1
met by current
ussed in [32],
GHz processor
es as long for
a dedicated
rocessor 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: EdDS

1985 ElGamal sigr
(R, S) is signature
if $B^{H(M)} \equiv A^R R^S$
and $R, S \in \{0, 1, .$

Here q is standard
B is standard base
A is signer's public
$H(M)$ is hash of n

Signer generates A
as secret powers o
easily solves for $S.$

. . .
eed
ly the
. . .

. . .

er
in 1
rrent
32],
ssor
for
d
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 $B^{H(M)} \equiv A^R R^S \quad (\mathrm{mod}\ q$
and $R, S \in \{0, 1, \ldots, q-2\}$

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

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

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

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

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

crypto" (CHES 2012)
z Cortex-A8 core:
les/byte (1.4 Gbps),
les/byte (3.4 Gbps)
n20, Poly1305.
cycles (2000/second),
cycles (1600/second)
e25519 DH, verify.

ortex-A8 was high-end
one core in 2010: e.g.,
g Exynos 3110 (Galaxy S);
P3630 (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 $R, S \in \{0, 1, \ldots, q-2\}$.

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

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

1990 Sc

1. Hash
$B^{H(M)} \equiv$
Reduces

2. Repl
with tw
$B^{H(M)/}$

Saves ti

3. Simp
$B^{H(M)/}$

Saves ti

4. Merg
$B^{H(R,M)}$
⇒ 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 $R, S \in \{0, 1, \ldots, q-2\}$.

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

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

1990 Schnorr impr

1. Hash R in the
$B^{H(M)} \equiv A^{H(R)} R$
Reduces attacker

2. Replace three e
with two exponent
$B^{H(M)/H(R)} \equiv AR$
Saves time in verif

3. Simplify by rela
$B^{H(M)/H(R)} \equiv AR$
Saves time in verif

4. Merge the hash
$B^{H(R,M)} \equiv AR^S$.
⇒ Resilient to H

2)

,

),

)

nd

g.,

axy S);

roid

ne 4).

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

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

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

1990 Schnorr improvements

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

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

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

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

Case study: EdDSA

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

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

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

1990 Schnorr improvements:

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

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

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

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

udy: EdDSA

Gamal signatures:

s signature of M

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

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

s standard prime,

ndard base,

ner's public key,

s hash of message.

enerates A and R

t powers of B;

lves for S.

1990 Schnorr improvements:

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

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

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

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

5. Elimi
$B^S \equiv R$
Simpler,

6. Comp
Saves sp

7. Use h
Saves sp

SA

natures:

e of M

(mod q)

$\ldots, q-2\}$.

l 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:
$B^{H(R,M)} \equiv AR^S$.
⇒ Resilient to H collisions.

---

5. Eliminate invers
$B^S \equiv RA^{H(R,M)}$.
Simpler, faster.

6. Compress R to
Saves space in sig

7. Use half-size H
Saves space in sig

1990 Schnorr improvements:

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

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

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

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

5. Eliminate inversions for s
$B^S \equiv R A^{H(R,M)}$.
Simpler, faster.

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

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

1990 Schnorr improvements:

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

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

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

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

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

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

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

1990 Schnorr improvements:

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

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

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

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

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

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

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

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

1990 Schnorr improvements:

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

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

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

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

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

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

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

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.

hnorr improvements:

R in the exponent:
$\equiv A^{H(R)}R^S$.
attacker control.

ace three exponents
o exponents:
$^{H(R)} \equiv AR^{S/H(R)}$.
me in verification.

lify by relabeling S:
$^{H(R)} \equiv AR^S$.
me in verification.

e the hashes:
$) \equiv AR^S$.
ient 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. ⇒ DSA, ECDSA avoided most improvements.

Patent expired in 2008.

EdDSA
Duif–La

Use ellip
−1-twist
⇒ very
natural s
no excep

Skip sign
Support

Use dou
and inclu

Generate
as a secr
⇒ Avoi

rovements:

exponent:
^{S}.

control.

exponents
ts:
$^{S/H(R)}$.

fication.

abeling S:
^{S}.

fication.

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. ⇒ DSA, ECDSA avoided most improvements.

Patent expired in 2008.

EdDSA (CHES 20
Duif–Lange–Schwa

Use elliptic curves
-1-twisted Edwar
⇒ very high spee
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
⇒ 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. ⇒ DSA, ECDSA avoided most improvements.

Patent expired in 2008.

EdDSA (CHES 2011 Bernst
Duif–Lange–Schwabe–Yang)

Use elliptic curves in “comp
−1-twisted Edwards” form.
⇒ very high speed,
natural side-channel protecti
no exceptional cases.

Skip signature compression.
Support batch verification.

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

Generate R deterministicall
as a secret hash of M.
⇒ Avoid PlayStation disast

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

Skip signature compression.
Support batch verification.

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

Generate R deterministically
as a secret hash of M.
⇒ Avoid PlayStation disaster.

nate inversions for signer:
$A^{H(R,M)}$.
faster.

press R to $H(R,M)$.
pace in signatures.

nalf-size H output.
pace in signatures.

ent research: extensive
cal study of security of
s system.

ented. ⇒ DSA, ECDSA
most improvements.

expired in 2008.

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

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

Skip signature compression.
Support batch verification.

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

Generate R deterministically
as a secret hash of M.
⇒ Avoid PlayStation disaster.

A higher

OpenSS
security
least sec
2.1 milli
on Corte

Our new
security
1.8 milli
on Corte

sions for signer:

$H(R, M)$.
natures.

output.
natures.

ch: extensive
f security of

DSA, ECDSA
rovements.

2008.

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

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

Skip signature compression.
Support batch verification.

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

Generate R deterministically
as a secret hash of M.
⇒ Avoid PlayStation disaster.

A higher-security

OpenSSL `secp16`
security level only
least secure ECC
2.1 million cycles
on Cortex-A8.

Our new Curve41
security level abov
1.8 million cycles
on Cortex-A8.

igner:

sive
of

DSA
.

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

Use elliptic curves in "complete -1-twisted Edwards" form.
⇒ very high speed,
natural side-channel protection,
no exceptional cases.

Skip signature compression.
Support batch verification.

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

Generate R deterministically
as a secret hash of M.
⇒ Avoid PlayStation disaster.

A higher-security curve

OpenSSL `secp160-r1`,
security level only 2^{80},
least secure ECC option:
2.1 million cycles
on Cortex-A8.

Our new Curve41417,
security level above 2^{200}:
1.8 million cycles
on Cortex-A8.

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.

A higher-security curve

OpenSSL `secp160-r1`,
security level only 2^{80},
least secure ECC option:
2.1 million cycles
on Cortex-A8.

Our new Curve41417,
security level above 2^{200}:
1.8 million cycles
on Cortex-A8.