High-Speed Cryptography

Peter Schwabe

National Taiwan University

Joint work with Daniel J. Bernstein, Tanja Lange

October 24, 2011

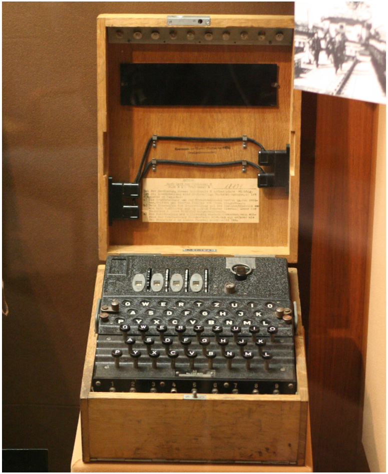
Graduate Seminar

Part I

Introduction to high-speed cryptography

The Enigma

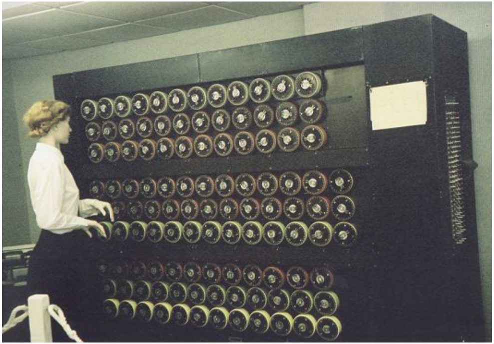
- Encryption device used by the German troops in WWII
- Developed by Scherbius, patented in 1928
- Variants with different number of rotors



Source: `http://en.wikipedia.org/wiki/File:Kriegsmarine_Enigma.png`, CC-by-sa-3.0

The Bombes

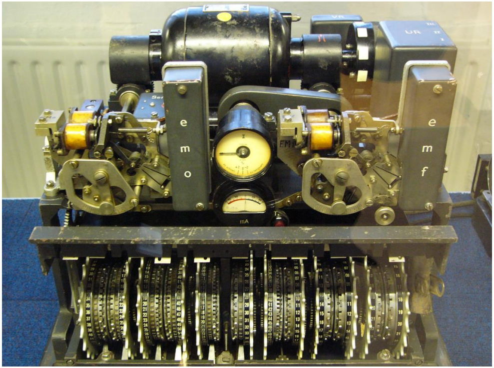
- Computing devices in Bletchley Park (UK)
- Used by the English to break the Enigma ciphers
- Large influence on the U-boat war



Source: `http://en.wikipedia.org/wiki/File:TuringBombeBletchleyPark.jpg`, GNU FDL 1.2

The Lorenz cipher machine

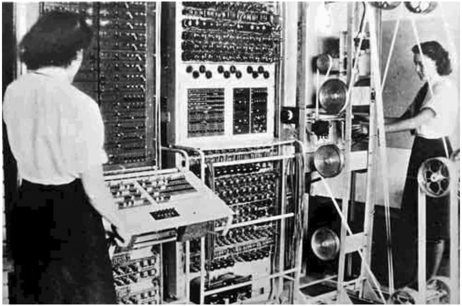
- Used by German army for high-level communication from ∼1942
- Extension to a Lorenz teleprinter
- Used a stream cipher



Source:
`http://en.wikipedia.org/wiki/File:Lorenz-SZ42-2.jpg`,
public domain

The Colossus

- First electronic digital information processing machine
- Used in Bletchley Park to break the Lorenz cipher from 1944



Source: `http://en.wikipedia.org/wiki/File:Colossus.jpg`, public domain

Computing and Cryptology

- All these machines can be seen as early "computers"
- The Bombes were developed by a team around Alan Turing, who is sometimes called "the inventor of the computer"

Computing and Cryptology

- All these machines can be seen as early "computers"
- The Bombes were developed by a team around Alan Turing, who is sometimes called "the inventor of the computer"
- Computers were built for cryptography, i.e. encryption (Enigma, Lorenz machine) ...

Computing and Cryptology

- All these machines can be seen as early "computers"
- The Bombes were developed by a team around Alan Turing, who is sometimes called "the inventor of the computer"
- Computers were built for cryptography, i.e. encryption (Enigma, Lorenz machine) ...
- ... or for cryptanalysis, i.e. breaking encryptions (Bombes, Colossus)

Computing and Cryptology

- All these machines can be seen as early "computers"
- The Bombes were developed by a team around Alan Turing, who is sometimes called "the inventor of the computer"
- Computers were built for cryptography, i.e. encryption (Enigma, Lorenz machine) ...
- ... or for cryptanalysis, i.e. breaking encryptions (Bombes, Colossus)
- Still today dedicated hardware is developed for encryption:
  - Various VIA processors feature the "PadLock Engine", hardware for the "Advanced Encryption Standard" (AES), hash algorithms, and more
  - Intel Processors since Westmere have built-in hardware support for AES (AES-NI instructions)
  - Even more common on embedded microprocessors to have hardware support for crypto

Computing and Cryptology II

- Reason for hardware support: Speed! (crypto needs to be fast)

Computing and Cryptology II

- Reason for hardware support: Speed! (crypto needs to be fast)
- Users don't want to experience a slowdown from, e.g., harddisk encryption

Computing and Cryptology II

- Reason for hardware support: Speed! (crypto needs to be fast)
- Users don't want to experience a slowdown from, e.g., harddisk encryption
- Faster harddisk encryption on laptops saves battery

Computing and Cryptology II

- Reason for hardware support: Speed! (crypto needs to be fast)
- Users don't want to experience a slowdown from, e.g., harddisk encryption
- Faster harddisk encryption on laptops saves battery
- Many servers spend most of their computation on encryption, faster crypto $\Rightarrow$ fewer servers, lower power bill, higher profit

Computing and Cryptology II

- Reason for hardware support: Speed! (crypto needs to be fast)
- Users don't want to experience a slowdown from, e.g., harddisk encryption
- Faster harddisk encryption on laptops saves battery
- Many servers spend most of their computation on encryption, faster crypto $\Rightarrow$ fewer servers, lower power bill, higher profit
- In principle this is true for all algorithms; cryptographic algorithms are "small", typically executed very often

Computing and Cryptology II

- Reason for hardware support: Speed! (crypto needs to be fast)
- Users don't want to experience a slowdown from, e.g., harddisk encryption
- Faster harddisk encryption on laptops saves battery
- Many servers spend most of their computation on encryption, faster crypto $\Rightarrow$ fewer servers, lower power bill, higher profit
- In principle this is true for all algorithms; cryptographic algorithms are "small", typically executed very often
- Obviously not all cryptographic algorithms supported by all processors in hardware
- Two effects:

Computing and Cryptology II

- Reason for hardware support: Speed! (crypto needs to be fast)
- Users don't want to experience a slowdown from, e.g., harddisk encryption
- Faster harddisk encryption on laptops saves battery
- Many servers spend most of their computation on encryption, faster crypto $\Rightarrow$ fewer servers, lower power bill, higher profit
- In principle this is true for all algorithms; cryptographic algorithms are "small", typically executed very often
- Obviously not all cryptographic algorithms supported by all processors in hardware
- Two effects:
  - Cryptographic algorithms are designed to be fast in software

Computing and Cryptology II

- Reason for hardware support: Speed! (crypto needs to be fast)
- Users don't want to experience a slowdown from, e.g., harddisk encryption
- Faster harddisk encryption on laptops saves battery
- Many servers spend most of their computation on encryption, faster crypto $\Rightarrow$ fewer servers, lower power bill, higher profit
- In principle this is true for all algorithms; cryptographic algorithms are "small", typically executed very often
- Obviously not all cryptographic algorithms supported by all processors in hardware
- Two effects:
  - Cryptographic algorithms are designed to be fast in software
  - Huge demand for high-speed software implementations of cryptography

Design for high-speed

- In 2000 NIST standardized Rijndael as AES, selection was between 5 algorithms
- Why did they not choose, e.g., Serpent? Let's see what NIST says:

Design for high-speed

- In 2000 NIST standardized Rijndael as AES, selection was between 5 algorithms
- Why did they not choose, e.g., Serpent? Let's see what NIST says:
- "Serpent appears to have a high security margin." ("Rijndael appears to have an adequate security margin.")

Design for high-speed

- In 2000 NIST standardized Rijndael as AES, selection was between 5 algorithms
- Why did they not choose, e.g., Serpent? Let's see what NIST says:
- "Serpent appears to have a high security margin." ("Rijndael appears to have an adequate security margin.")
- "Serpent is well suited to restricted-space environments"

Design for high-speed

- In 2000 NIST standardized Rijndael as AES, selection was between 5 algorithms
- Why did they not choose, e.g., Serpent? Let's see what NIST says:
- "Serpent appears to have a high security margin." ("Rijndael appears to have an adequate security margin.")
- "Serpent is well suited to restricted-space environments"
- "[Hardware] Efficiency is generally very good"

Design for high-speed

- In 2000 NIST standardized Rijndael as AES, selection was between 5 algorithms
- Why did they not choose, e.g., Serpent? Let's see what NIST says:
- "Serpent appears to have a high security margin." ("Rijndael appears to have an adequate security margin.")
- "Serpent is well suited to restricted-space environments"
- "[Hardware] Efficiency is generally very good"
- "Serpent is generally the slowest of the finalists in software speed for encryption and decryption"

Design for high-speed

- In 2000 NIST standardized Rijndael as AES, selection was between 5 algorithms
- Why did they not choose, e.g., Serpent? Let's see what NIST says:
- "Serpent appears to have a high security margin." ("Rijndael appears to have an adequate security margin.")
- "Serpent is well suited to restricted-space environments"
- "[Hardware] Efficiency is generally very good"
- "Serpent is generally the slowest of the finalists in software speed for encryption and decryption"
- Similar for currently running SHA-3 competition: software speed one of the most important selection criteria

High-speed software implementations

- Very common to implement algorithms in assembly
- A 20% speedup is often worth a publication
- Workshop entirely on this topic: "SPEED – Software Performance Enhancement for Encryption and Decryption"

High-speed software implementations

- Very common to implement algorithms in assembly
- A 20% speedup is often worth a publication
- Workshop entirely on this topic: "SPEED – Software Performance Enhancement for Encryption and Decryption"

Definition

The term *high-speed cryptography* means the design and implementation of secure and fast cryptographic software for off-the-shelf computers.

What high-speed crypto is *not*
(at least not in this talk)

- Design of cryptographic primitives targeting high performance
- Implementing crypto in hardware
- Making crypto faster by choosing low-security *functions*
- Making crypto faster by low-security *implementations*

Writing secure and fast cryptographic software

- High-level parameter choices (mathematical structures, e.g., finite fields, elliptic curves)

Writing secure and fast cryptographic software

- High-level parameter choices (mathematical structures, e.g., finite fields, elliptic curves)
- Choice of high-level algorithms (e.g., scalar multiplication, exponentiation)

Writing secure and fast cryptographic software

- High-level parameter choices (mathematical structures, e.g., finite fields, elliptic curves)
- Choice of high-level algorithms (e.g., scalar multiplication, exponentiation)
- Representation of structures and low-level algorithms (e.g., representation of big integers, modular multiplication algorithm, bitslicing technique)

Writing secure and fast cryptographic software

- High-level parameter choices (mathematical structures, e.g., finite fields, elliptic curves)
- Choice of high-level algorithms (e.g., scalar multiplication, exponentiation)
- Representation of structures and low-level algorithms (e.g., representation of big integers, modular multiplication algorithm, bitslicing technique)
- Careful optimization on the assembly level
  - Fast software
  - Secure software

Writing secure and fast cryptographic software

- High-level parameter choices (mathematical structures, e.g., finite fields, elliptic curves)
- Choice of high-level algorithms (e.g., scalar multiplication, exponentiation)
- Representation of structures and low-level algorithms (e.g., representation of big integers, modular multiplication algorithm, bitslicing technique)
- Careful optimization on the assembly level
  - Fast software
  - Secure software
- Considerations of subtle interactions between these levels (e.g., a certain set of high-level parameters may only be "good" for certain microarchitectures)

Secure software implementations

- Just because a cryptographic function is considered secure, an *implementation* of this function can still be insecure

Secure software implementations

- Just because a cryptographic function is considered secure, an *implementation* of this function can still be insecure
- Example 1:

```
if(secretbit)
  f();
else
  g();
```

- This piece of code takes a different amount of time, depending on the value of `secretbit`
- Opens the door for a *timing attack*: Attacker measures the time, draws conclusions about secret data (e.g., the key)

Secure software implementations

- Just because a cryptographic function is considered secure, an *implementation* of this function can still be insecure
- Example 1:

```
if(secretbit)
  f();
else
  g();
```

- This piece of code takes a different amount of time, depending on the value of `secretbit`
- Opens the door for a *timing attack*: Attacker measures the time, draws conclusions about secret data (e.g., the key)
- Example 2:

```
x = lookuptable[secret_position];
```

Secure software implementations

- Just because a cryptographic function is considered secure, an *implementation* of this function can still be insecure
- Example 1:

```
if(secretbit)
  f();
else
  g();
```

- This piece of code takes a different amount of time, depending on the value of `secretbit`
- Opens the door for a *timing attack*: Attacker measures the time, draws conclusions about secret data (e.g., the key)
- Example 2:

```
x = lookuptable[secret_position];
```

- This code takes different amount of time, depending on whether the table entry at `secure_position` is in cache or not

Secure software implementations

- Just because a cryptographic function is considered secure, an *implementation* of this function can still be insecure
- Example 1:

```
if(secretbit)
  f();
else
  g();
```

- This piece of code takes a different amount of time, depending on the value of `secretbit`
- Opens the door for a *timing attack*: Attacker measures the time, draws conclusions about secret data (e.g., the key)
- Example 2:

```
x = lookuptable[secret_position];
```

- This code takes different amount of time, depending on whether the table entry at `secure_position` is in cache or not
- Again: The attacker can influence the cache, measure time...

Part II

The security impact of a new cryptographic library

Crypto software state of the art

- Well studied and understood cryptographic algorithms (AES, SHA-256, RSA-2048)
- Breaking these algorithms considered infeasible
- Various implementations available in public cryptographic libraries (e.g., OpenSSL)
- Common best practice: Use these libraries

Crypto software state of the art

- Well studied and understood cryptographic algorithms (AES, SHA-256, RSA-2048)
- Breaking these algorithms considered infeasible
- Various implementations available in public cryptographic libraries (e.g., OpenSSL)
- Common best practice: Use these libraries
- Cryptography is still a disaster, many complete failures of confidentiality and integrity

The NaCl library

- We designed and implemented a new cryptographic library: NaCl
- Stands for "Networking and Cryptography library", pronounced "salt"
- Acknowledgements: Code contributions from Matthew Dempsky (Mochi Media), Niels Duif (TU Eindhoven), Emilia Käsper (KU Leuven, now Google), Adam Langley (Google), Bo-Yin Yang (Academia Sinica)

Goal of NaCl

- Most of Internet traffic is not cryptographically protected
- Main goal of NaCl: Change this!

Goal of NaCl

- Most of Internet traffic is not cryptographically protected
- Main goal of NaCl: Change this!
- Alice has a message m for Bob
- Use Bob's public key and Alice's private key to compute authenticated ciphertext c
- Send c to Bob

Goal of NaCl

- Most of Internet traffic is not cryptographically protected
- Main goal of NaCl: Change this!
- Alice has a message m for Bob
- Use Bob's public key and Alice's private key to compute authenticated ciphertext c
- Send c to Bob
- Bob uses Alice's public key and his private 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 private key from wire format
- Use key to sign hash
- Read Bob's public key from wire format
- Use key to encrypt AES key, signature etc.
- Convert to wire format

Alice using a typical cryptographic library

- Generate random AES key
- Use AES key to encrypt packet
- Hash encrypted packet
- Read RSA private key from wire format
- Use key to sign hash
- Read Bob's public key from wire format
- Use key to encrypt AES key, signature etc.
- Convert to wire format
- Plus more code: allocate storage, handle errors etc.

Alice using NaCl

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

Alice using NaCl

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

- 32-byte private key `sk`
- 32-byte public key `pk`
- 24-byte nonce `n`
- message `m`

Alice using NaCl

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

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

Alice using NaCl

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

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

Alice using NaCl

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

- 32-byte private key `sk`
- 32-byte public key `pk`
- 24-byte nonce `n`
- message `m`
- `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 using NaCl

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

Bob using NaCl

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

- Initial key-pair generation:

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

Signatures in NaCl

- Can (instead) use **signatures** for public messages:

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

- 64-byte private key `sk`
- 32-byte public key `pk`

Signatures in NaCl

- Can (instead) use **signatures** for public messages:

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

- 64-byte private key `sk`
- 32-byte public key `pk`
- Signing:

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

Signatures in NaCl

- Can (instead) use **signatures** for public messages:

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

- 64-byte private key `sk`
- 32-byte public key `pk`
- Signing:

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

- Verification

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

No secret load addresses

- 2005 paper by Osvik, Shamir, Tromer: 65 ms to steal Linux AES key used for hard-disk encryption (`dm-crypt`)
- Attack needs a process on the same CPU, but without privileges

No secret load addresses

- 2005 paper by Osvik, Shamir, Tromer: 65 ms to steal Linux AES key used for hard-disk encryption (`dm-crypt`)
- Attack needs a process on the same CPU, but without privileges
- Almost all AES implementations use fast lookup tables
- Highly vulnerable to cache-timing attacks

No secret load addresses

- 2005 paper by Osvik, Shamir, Tromer: 65 ms to steal Linux AES key used for hard-disk encryption (`dm-crypt`)
- Attack needs a process on the same CPU, but without privileges
- Almost all AES implementations use fast lookup tables
- Highly vulnerable to cache-timing attacks
- Most cryptographic libraries still use lookup tables, but add "countermeasures"
- Not confidence-inspiring, likely to be breakable

No secret load addresses

- 2005 paper by Osvik, Shamir, Tromer: 65 ms to steal Linux AES key used for hard-disk encryption (`dm-crypt`)
- Attack needs a process on the same CPU, but without privileges
- Almost all AES implementations use fast lookup tables
- Highly vulnerable to cache-timing attacks
- Most cryptographic libraries still use lookup tables, but add "countermeasures"
- Not confidence-inspiring, likely to be breakable
- NaCl systematically avoids *all* loads from addresses that depend on secret data
- `ctgrind` (2010 by Langley): tool to validate this automatically

No secret branch conditions

- 2011 paper by Brumley, Tuveri: minutes to steal another machine's OpenSSL ECDSA key
- Attack exploits timing variation from secret branch conditions
- Most cryptographic libraries have many small-scale variations in timing, e.g. from `memcmp`
- NaCl systematically avoids *all* branch conditions that depend on secret data

No padding oracles

- 1998 paper by Bleichenbacher: Decrypt SSL RSA ciphertext
- Attack observes server responses to $\approx 10^6$ variants of forged ciphertext

No padding oracles

- 1998 paper by Bleichenbacher: Decrypt SSL RSA ciphertext
- Attack observes server responses to $\approx 10^6$ variants of forged ciphertext
- SSL first inverts RSA, then checks for "PKCS padding"
- Server reponses reveal pattern of PKCS forgeries, pattern reveals plaintext

No padding oracles

- 1998 paper by Bleichenbacher: Decrypt SSL RSA ciphertext
- Attack observes server responses to $\approx 10^6$ variants of forged ciphertext
- SSL first inverts RSA, then checks for "PKCS padding"
- Server reponses reveal pattern of PKCS forgeries, pattern reveals plaintext
- Typical defense: Try to hide differences between padding checks and subsequent integrity checks
- Hard to get this right: 2009 paper by Albrecht, Paterson, Watson recovered some SSH plaintext

No padding oracles

- 1998 paper by Bleichenbacher: Decrypt SSL RSA ciphertext
- Attack observes server responses to $\approx 10^6$ variants of forged ciphertext
- SSL first inverts RSA, then checks for "PKCS padding"
- Server reponses reveal pattern of PKCS forgeries, pattern reveals plaintext
- Typical defense: Try to hide differences between padding checks and subsequent integrity checks
- Hard to get this right: 2009 paper by Albrecht, Paterson, Watson recovered some SSH plaintext
- NaCl does not decrypt unless message is authenticated
- Verification rejects forgeries in constant time

Centralizing randomness

- Observation by Bello (2008): Debian/Ubuntu OpenSSL keys had only 15 bits of entropy for 1.5 years
- Attacker could just try all 32768 possible keys

Centralizing randomness

- Observation by Bello (2008): Debian/Ubuntu OpenSSL keys had only 15 bits of entropy for 1.5 years
- Attacker could just try all 32768 possible keys
- Huge effort to blacklist all insecure keys, generate and deploy new keys

Centralizing randomness

- Observation by Bello (2008): Debian/Ubuntu OpenSSL keys had only 15 bits of entropy for 1.5 years
- Attacker could just try all 32768 possible keys
- Huge effort to blacklist all insecure keys, generate and deploy new keys
- Problem was: Debian developer had removed a subtle line of OpenSSL randomness-generating code

Centralizing randomness

- Observation by Bello (2008): Debian/Ubuntu OpenSSL keys had only 15 bits of entropy for 1.5 years
- Attacker could just try all 32768 possible keys
- Huge effort to blacklist all insecure keys, generate and deploy new keys
- Problem was: Debian developer had removed a subtle line of OpenSSL randomness-generating code
- NaCl retrieves all randomness from `/dev/urandom`, the OS random-number generator
- Reviewing this code is much more tractable than reviewing RNG code in every security library

Avoiding unnecessary randomness

- ECDSA signatures require new randomness for each signature
- Sony ignored this requirement for PS3 code signing
- 2010 presentation by Bushing, Marcan, Segher, Sven: Complete break of the PS3 security system

Avoiding unnecessary randomness

- ECDSA signatures require new randomness for each signature
- Sony ignored this requirement for PS3 code signing
- 2010 presentation by Bushing, Marcan, Segher, Sven: Complete break of the PS3 security system
- NaCl has deterministic `crypto_box` and `crypto_sign`
- Randomness is only required for `keypair` functions

Avoiding unnecessary randomness

- ECDSA signatures require new randomness for each signature
- Sony ignored this requirement for PS3 code signing
- 2010 presentation by Bushing, Marcan, Segher, Sven: Complete break of the PS3 security system
- NaCl has deterministic `crypto_box` and `crypto_sign`
- Randomness is only required for `keypair` functions
- Eliminates this kind of disaster
- Also simplifies testing

Avoiding pure crypto failures

- In 2008 Stevens, Sotirov, Appelbaum, Lenstra, Molnar, Osvik, de Weger exploited MD5 weakness to create a rogue CA certificate
- Such certificates can be used to impersonate any `https` website

Avoiding pure crypto failures

- In 2008 Stevens, Sotirov, Appelbaum, Lenstra, Molnar, Osvik, de Weger exploited MD5 weakness to create a rogue CA certificate
- Such certificates can be used to impersonate any `https` website
- Already in 1996 Preneel and Dobbertin called for MD5 to be scrapped

Avoiding pure crypto failures

- In 2008 Stevens, Sotirov, Appelbaum, Lenstra, Molnar, Osvik, de Weger exploited MD5 weakness to create a rogue CA certificate
- Such certificates can be used to impersonate any `https` website
- Already in 1996 Preneel and Dobbertin called for MD5 to be scrapped
- NaCl pays attention to cryptanalysis and makes very conservative choices of cryptographic primitives

Speed

- Crypto performance problems lead to users reducing security levels or giving up on crypto

Speed

- Crypto performance problems lead to users reducing security levels or giving up on crypto
  - Example 1: Google SSL uses RSA-1024
  - Analysis in 2003 concluded that RSA-1024 was breakable
  - Shamir-Tromer estimated 1 year, $\approx 10^7$ USD.

Speed

- Crypto performance problems lead to users reducing security levels or giving up on crypto
  - Example 1: Google SSL uses RSA-1024
  - Analysis in 2003 concluded that RSA-1024 was breakable
  - Shamir-Tromer estimated 1 year, $\approx 10^7$ USD.
  - Example 2: Tor anonymizer uses RSA-1024

Speed

- Crypto performance problems lead to users reducing security levels or giving up on crypto
  - Example 1: Google SSL uses RSA-1024
  - Analysis in 2003 concluded that RSA-1024 was breakable
  - Shamir-Tromer estimated 1 year, $\approx 10^7$ USD.
  - Example 2: Tor anonymizer uses RSA-1024
  - Example 3: DNSSEC uses RSA-1024 "tradeoff between the risk of key compromise and performance"

Speed

- Crypto performance problems lead to users reducing security levels or giving up on crypto
  - Example 1: Google SSL uses RSA-1024
  - Analysis in 2003 concluded that RSA-1024 was breakable
  - Shamir-Tromer estimated 1 year, $\approx 10^7$ USD.
  - Example 2: Tor anonymizer uses RSA-1024
  - Example 3: DNSSEC uses RSA-1024 "tradeoff between the risk of key compromise and performance"
  - Example 4: https://sourceforge.net/account is proteced by SSL, but https://sourceforge.net/develop redirects to http://sourceforge.net/develop, turning of cryptography

Speed

- Crypto performance problems lead to users reducing security levels or giving up on crypto
  - Example 1: Google SSL uses RSA-1024
  - Analysis in 2003 concluded that RSA-1024 was breakable
  - Shamir-Tromer estimated 1 year, $\approx 10^7$ USD.
  - Example 2: Tor anonymizer uses RSA-1024
  - Example 3: DNSSEC uses RSA-1024 “tradeoff between the risk of key compromise and performance”
  - Example 4: `https://sourceforge.net/account` is proteced by SSL, but `https://sourceforge.net/develop` redirects to `http://sourceforge.net/develop`, turning of cryptography
- NaCl has no low-security options:
  - `crypto_box` always encrypts *and* authenticates
  - no RSA-1024, not even RSA-2048

NaCl speed

- NaCl is exceptionally fast, much faster than other libraries
- *Keeps up with the network*

NaCl speed

- NaCl is exceptionally fast, much faster than other libraries
- *Keeps up with the network*
- Operations per second on an AMD Phenom II X6 1100 T (164 €)
  - `crypto_box`: More than 80000
  - `crypto_box_open`: More than 80000
  - `crypto_sign_open`: More than 70000
  - `crypto_sign`: More than 180000

NaCl speed

- NaCl is exceptionally fast, much faster than other libraries
- *Keeps up with the network*
- Operations per second on an AMD Phenom II X6 1100 T (164 €)
  - `crypto_box`: More than 80000
  - `crypto_box_open`: More than 80000
  - `crypto_sign_open`: More than 70000
  - `crypto_sign`: More than 180000
- 80000 1500-byte packets/second fill up a 1Gbps link

Even more NaCl speed

- Many packets to the same public key can gain speed: Split `crypto_box` into `crypto_box_beforenm` and `crypto_box_afternm`
- Perform operations depending only on the keys `sk` and `pk` only once (in `crypto_box_beforenm`)

Even more NaCl speed

- Many packets to the same public key can gain speed: Split `crypto_box` into `crypto_box_beforenm` and `crypto_box_afternm`
- Perform operations depending only on the keys `sk` and `pk` only once (in `crypto_box_beforenm`)
- Batch verification for signatures: double verification speed for a batch of 64 valid signatures

More information

NaCl Website: http://nacl.cr.yp.to
All code is in the public domain: Use it any way you want!

More information

NaCl Website: http://nacl.cr.yp.to
All code is in the public domain: Use it any way you want!

Paper "*The security impact of a new cryptographic library*" will be online soon at http://cryptojedi.org/papers/#coolnacl