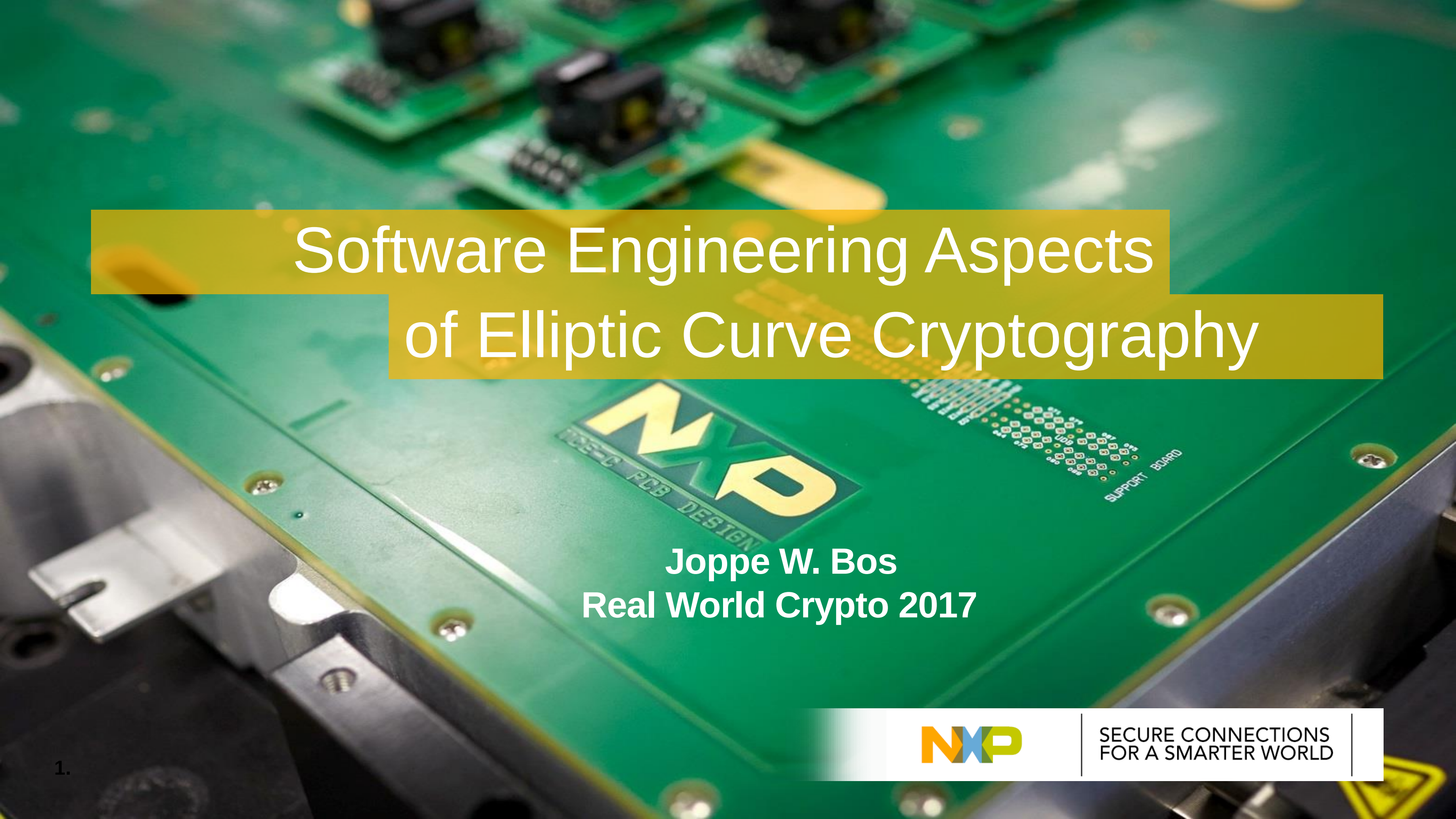


Software Engineering Aspects of Elliptic Curve Cryptography

Joppe W. Bos

Real World Crypto 2017

SECURE CONNECTIONS FOR A SMARTER WORLD

NXP Semiconductors

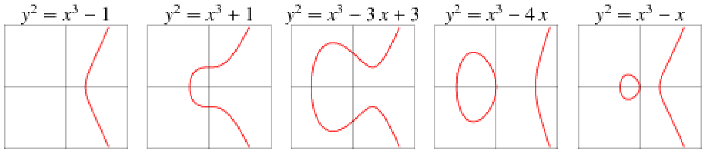
Operations in > 35 countries, more than 130 facilities
≈ 45,000 employees

Research & Development
≈ 11,200 engineers in 23 countries

Elliptic Curves

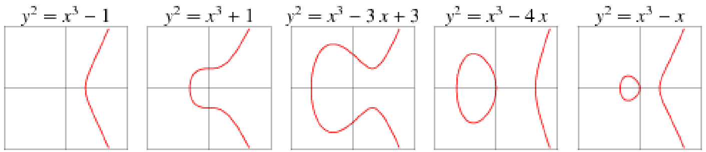
What is an elliptic curve?

Not an ellipse!



Elliptic Curves

What is an elliptic curve?



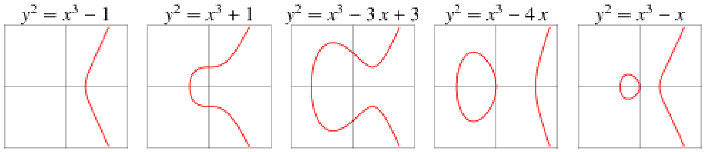
Not an ellipse!

<u>Mathematical perspective</u>

Smooth, projective algebraic curve of genus one which together with a point “at infinity” forms an abelian variety

Elliptic Curves

What is an elliptic curve?



Not an ellipse!

Mathematical perspective

Smooth, projective algebraic curve of genus one which together with a point “at infinity” forms an abelian variety

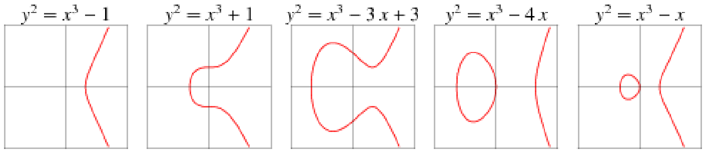
Practical perspective

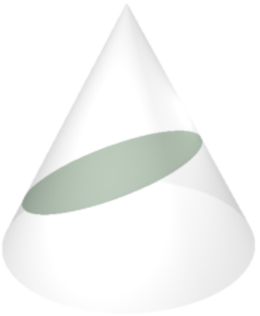
When defined over a large prime field an elliptic curve simply is

$$E/\mathbb{F}_p : y^2 = x^3 + ax + b \quad \text{such that} \quad 4a^3 + 27b^2 \neq 0$$

Elliptic Curves

What is an elliptic curve?





Not an ellipse!

Mathematical perspective
Smooth, projective algebraic curve of genus one which together with a point “at infinity” forms an abelian variety

Practical perspective
When defined over a large prime field an elliptic curve simply is

$$E/\mathbb{F}_p : y^2 = x^3 + ax + b \qquad \text{such that} \qquad 4a^3 + 27b^2 \neq 0$$

Engineering perspective
An “algorithm” which needs to be implemented in a “secure” way

Goal of this Lecture

- Creating ECC implementations is **easy**
  - Play around with Sage, Magma
  - Even in C this is trivial

Goal of this Lecture

- Creating ECC implementations is **easy**
  - Play around with Sage, Magma
  - Even in C this is trivial
- Creating efficient (performance / memory / binary size) ECC implementations is **a challenge**

Goal of this Lecture

- Creating ECC implementations is **easy**
  - Play around with Sage, Magma
  - Even in C this is trivial
- Creating efficient (performance / memory / binary size) ECC implementations is **a challenge**
- Creating efficient and secure ECC implementations is **hard**
  - Define “secure”?

Goal.

- Show some examples how different settings of “secure” have an impact on ECC software design in practice.
- Common mistakes made in practice.

ECC in Practice
Security 101

SECURE CONNECTIONS
FOR A SMARTER WORLD

Elliptic Curves in Hardware and Software in Practice

We see an increase in support for ECC in software, for example

- 2013 scan observed: “about 1 in 10 systems support ECC across the TLS and SSH protocols”
- Around **5 million hosts** support ECC in TLS / SSH
- Many TLS servers prefer ciphersuites with ECDHE

Elliptic Curves in Hardware and Software in Practice

We see an increase in support for ECC in software, for example

- 2013 scan observed: “about 1 in 10 systems support ECC across the TLS and SSH protocols”
- Around **5 million hosts** support ECC in TLS / SSH
- Many TLS servers prefer ciphersuites with ECDHE

Hardware ECC

- ✓ Currently, ECC coprocessors are used
  - ✓ in **billions** of smart cards securing ID cards, passports and banking
  - ✓ for 15 years in devices supporting the Digital Transmission Content Protection system

(Short-term) future: Internet-of-Things, prediction

- ✓ 5 billion things at the end of 2015
- ✓ 25 billion things around 2020

- For asymmetric crypto, ECC is the logical choice: small keys, fast on embedded platforms, etc
- Many “things” need to communicate securely with user-apps and possibly the world wide web
- Hardware and software implementation will start to talk to each other (more frequently)!

ECC Keys

Domain parameters

$$(p, a, b, G, n, h)$$

- $p \in \mathbb{Z}$ prime number which defines $\mathbb{F}_p$
- $a, b \in \mathbb{F}_p$ define $y^2 = x^3 + ax + b$
- $G = (x, y) \in E(\mathbb{F}_p)$
- $n \in \mathbb{Z}$ prime order of G
- $h \in \mathbb{Z}$ co-factor, $\mathrm{h} = \#E(\mathbb{F}_p)/n$

ECC Keys

Private key: $d \in \mathbb{Z}/n\mathbb{Z}$
Public key: $P = d \cdot G \in E(\mathbb{F}_p)$

<u>Domain parameters</u>

$$(p, a, b, G, n, h)$$

- $p \in \mathbb{Z}$ prime number which defines $\mathbb{F}_p$
- $a, b \in \mathbb{F}_p$ define $y^2 = x^3 + ax + b$
- $G = (x, y) \in E(\mathbb{F}_p)$
- $n \in \mathbb{Z}$ prime order of G
- $h \in \mathbb{Z}$ co-factor, $\mathrm{h} = \#E(\mathbb{F}_p)/n$

ECC Keys

Domain parameters

$$(p, a, b, G, n, h)$$

- $p \in \mathbb{Z}$ prime number which defines $\mathbb{F}_p$
- $a, b \in \mathbb{F}_p$ define $y^2 = x^3 + ax + b$
- $G = (x, y) \in E(\mathbb{F}_p)$
- $n \in \mathbb{Z}$ prime order of G
- $h \in \mathbb{Z}$ co-factor, $\mathrm{h} = \#E(\mathbb{F}_p)/n$

These domain parameters are publicly available through named identifiers

Private key: $d \in \mathbb{Z}/n\mathbb{Z}$
Public key: $P = d \cdot G \in E(\mathbb{F}_p)$

NIST	SEC	ANSI X9.62	OpenSSL
Curve P-192	secp192r1	prime192v1	prime192v1
Curve P-224	secp224r1		secp224r1
Curve P-256	secp256r1	prime256v1	prime256v1
Curve P-384	secp384r1		secp384r1
Curve P-521	secp521r1		secp521r1

Programming 101

Low level: The implementation → the basics

```
static int buffer[128];

int read_buffer(int index) {
  if (index < 128)
    return buffer[index];
  return ERROR;
}
```

What is wrong with this code?

Programming 101

Low level: The implementation → the basics

```
static int buffer[128];

int read_buffer(int index) {
  if (index < 128)
    return buffer[index];
  return ERROR;
}
```

What is wrong with this code?

Buffer underrun!

Programming 101

```
static int buffer[128];

int read_buffer(int index) {
  if (index < 128)
    return buffer[index];
  return ERROR;
}
```

What is wrong with this code?

Buffer underrun!

Since C has been used for more than 30 years resulting in a large base of legacy code that is still being used in present day (new) products.

Much of the legacy code dates back from even before the C language standardization.

Legacy code requires significantly more effort to secure than more recent code due to :

- Coding style
- Lack of security knowledge during implementation
- Loose compiler standards at the time of implementation

ANSI-C offers by default little to no security measures

ECDSA

Alice´s private signature key

Alice´s public verification key

sk_A vk_A

(m, s)

$S(m, sk_A) = s \longrightarrow V(m, s, vk_A) = true/false$

signature generation function

message and signature

signature verification function

High level: The protocol → the basics

ECDSA

Signature generation

Def $(r, s) = \text{sign}(m)$ {
  Repeat {
    Repeat {
      Select random $k \in [1, \ldots n-1]$
      Compute $k \cdot P = (x, y)$
      Compute $r = x \bmod n$
    } until $(r \neq 0)$
    Compute $e = \mathcal{H}(m)$
    Compute $s = k^{-1}(e + dr) \bmod n$
  } until $(s \neq 0)$
  Return (r, s)
}

ECDSA

Signature generation

Def $(r, s) = \text{sign}(m)$ {
  Repeat {
    Repeat {
      Select random $k \in [1, \ldots n-1]$
      Compute $k \cdot P = (x, y)$
      Compute $r = x \bmod n$
    } until $(r \neq 0)$
    Compute $e = \mathcal{H}(m)$
    Compute $s = k^{-1}(e + dr) \bmod n$
  } until $(s \neq 0)$
  Return (r, s)
}

Signature verification

Def {accept,reject} $= \text{verify}(r, s)$ {
  If ($r < 0$ or $r \geq n$ or $s < 0$ or $s \geq n$) return reject
  Compute $e = \mathcal{H}(m)$
  Compute $w = s^{-1} \bmod n$
  Compute $u_1 = ew \bmod n$ and $u_2 = rw \bmod n$
  Compute $X = u_1 \cdot P + u_2 \cdot Q = (x, y)$
  If ($X == \mathcal{O}$) return reject
  If ($x \bmod n \neq r$) return reject
  Return accept
}

ECDSA

Signature generation

Def $(r, s) = \text{sign}(m)$ {
  Repeat {
    Repeat {
      Select random $k \in [1, \dots n-1]$
      Compute $k \cdot P = (x, y)$
      Compute $r = x \bmod n$
    } until $(r \neq 0)$
    Compute $e = \mathcal{H}(m)$
    Compute $s = k^{-1}(e + dr) \bmod n$
  } until $(s \neq 0)$
  Return (r, s)
}

Signature verification

Def {accept,reject} $= \text{verify}(r, s)$ {
  If ($r < 0$ or $r \geq n$ or $s < 0$ or $s \geq n$) return reject
  Compute $e = \mathcal{H}(m)$
  Compute $w = s^{-1} \bmod n$
  Compute $u_1 = ew \bmod n$ and $u_2 = rw \bmod n$
  Compute $X = u_1 \cdot P + u_2 \cdot Q = (x, y)$
  If ($X == \mathcal{O}$) return reject
  If ($x \bmod n \neq r$) return reject
  Return accept
}

$$s = k^{-1}(e + dr) \rightarrow k \equiv s^{-1}e + s^{-1}dr \equiv we + wrd \equiv u_1 + u_2 d \pmod{n}$$

ECDSA

Signature generation

Def $(r, s) = \text{sign}(m)$ {
  Repeat {
    Repeat {
      Select random $k \in [1, \ldots n-1]$
      Compute $k \cdot P = (x, y)$
      Compute $r = x \bmod n$
    } until $(r \neq 0)$
    Compute $e = \mathcal{H}(m)$
    Compute $s = k^{-1}(e + dr) \bmod n$
  } until $(s \neq 0)$
  Return (r, s)
}

Signature verification

Def {accept,reject} $= \text{verify}(r, s)$ {
  If ($r < 0$ or $r \geq n$ or s < 0 or s $\geq n$) return reject
  Compute $e = \mathcal{H}(m)$
  Compute $w = s^{-1} \bmod n$
  Compute $u_1 = ew \bmod n$ and $u_2 = rw \bmod n$
  Compute $X = u_1 \cdot P + u_2 \cdot Q = (x, y)$
  If ($X == \mathcal{O}$) return reject
  If ($x \bmod n \neq r$) return reject
  Return accept
}

$$s = k^{-1}(e + dr) \rightarrow k \equiv s^{-1}e + s^{-1}dr \equiv we + wrd \equiv u_1 + u_2 d \pmod{n}$$

$$X = u_1 P + u_2 Q = (u_1 + u_2 d)P = kP \rightarrow x \bmod n = r$$

ECDSA – Security 101

The value r has **the same security requirements** as the private key d

Using the same random $k \rightarrow kP = (x, y) \rightarrow r = x \bmod n$ is also the same	
$\text{Sign}(m_1) = (r, s_1)$	$\text{Sign}(m_2) = (r, s_2)$

ECDSA – Security 101

The value r has **the same security requirements** as the private key d

Using the same random $k \rightarrow kP = (x, y) \rightarrow r = x \bmod n$ is also the same	
$\text{Sign}(m_1) = (r, s_1)$	$\text{Sign}(m_2) = (r, s_2)$
$e_1 = \mathcal{H}(m_1)$	$e_2 = \mathcal{H}(m_2)$
$s_1 = k^{-1}(e_1 + d \cdot r) \bmod n$	$s_2 = k^{-1}(e_2 + d \cdot r) \bmod n$

ECDSA – Security 101

The value r has **the same security requirements** as the private key d

Using the same random $k \rightarrow kP = (x, y) \rightarrow r = x \bmod n$ is also the same	
$\text{Sign}(m_1) = (r, s_1)$	$\text{Sign}(m_2) = (r, s_2)$
$e_1 = \mathcal{H}(m_1)$	$e_2 = \mathcal{H}(m_2)$
$s_1 = k^{-1}(e_1 + d \cdot r) \bmod n$	$s_2 = k^{-1}(e_2 + d \cdot r) \bmod n$
$\mathrm{k} \cdot s_1 = e_1 + d \cdot r \bmod n$	$\mathrm{k} \cdot s_2 = e_2 + d \cdot r \bmod n$
$\mathrm{k} \cdot (s_1 - s_2) \equiv e_1 - e_2 \bmod n \quad \rightarrow \quad k \equiv (e_1 - e_2) \cdot (s_1 - s_2)^{-1} \bmod n$	

We can compute k

ECDSA – Security 101

The value r has **the same security requirements** as the private key d

Using the same random $k \to kP = (x, y) \to r = x \bmod n$ is also the same	
$\text{Sign}(m_1) = (r, s_1)$	$\text{Sign}(m_2) = (r, s_2)$
$e_1 = \mathcal{H}(m_1)$	$e_2 = \mathcal{H}(m_2)$
$s_1 = k^{-1}(e_1 + d \cdot r) \bmod n$	$s_2 = k^{-1}(e_2 + d \cdot r) \bmod n$
$\text{k} \cdot s_1 = e_1 + d \cdot r \bmod n$	$\text{k} \cdot s_2 = e_2 + d \cdot r \bmod n$
$\text{k} \cdot (s_1 - s_2) \equiv e_1 - e_2 \bmod n \quad \to \quad k \equiv (e_1 - e_2) \cdot (s_1 - s_2)^{-1} \bmod n$	
$s = k^{-1}(e_1 + d \cdot r) \quad \to \quad d = r^{-1}(k \cdot s - e_1) \bmod n$	

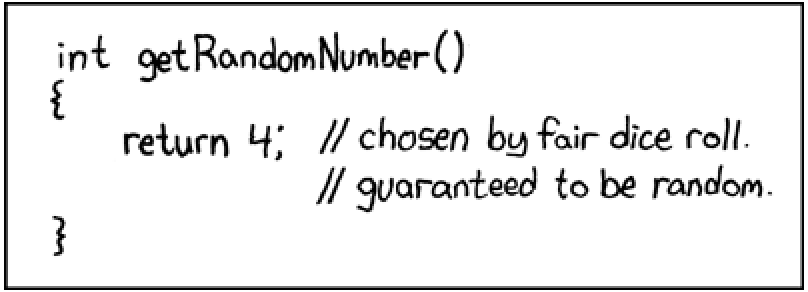
We can compute k, which allows us to compute the secret key d

ECDSA – Security 101

Nobody would hard-code this random value k right?

ECDSA – Security 101

Nobody would hard-code this random value k right?



Terrible example

Used in 2010 to get the private key from Sony's video game console PlayStation 3.
The per-message random value k was hard-coded.

Fast Scalar Multiplications

SECURE CONNECTIONS FOR A SMARTER WORLD

Elliptic Curve Scalar Multiplication

In ECDSA and ECDH(E) the scalar multiplication is the most time consuming

Input: $G \in E(\mathbb{F}_p)$ and $\mathbb{Z} \ni s = \sum_{i=0}^{k-1} s_i \cdot 2^i$

Output: $s \cdot G \in E(\mathbb{F}_p)$

1. $P \leftarrow G$
2. for ($i = k - 2$; $i \geq 0$; i--) {
3. $\quad P \leftarrow 2 \cdot P$ (double)
4. $\quad$ if ($s_i == 1$) $P \leftarrow P + G$ (add)
5. }
6. Return P

Elliptic Curve Scalar Multiplication

In ECDSA and ECDH(E) the scalar multiplication is the most time consuming

Input: $G \in E(\mathbb{F}_p)$ and $\mathbb{Z} \ni s = \sum_{i=0}^{k-1} s_i \cdot 2^i$
Output: $s \cdot G \in E(\mathbb{F}_p)$
1. $P \leftarrow G$
2. for ($i = k - 2$; $i \geq 0$; i--) {
3. $\quad P \leftarrow 2 \cdot P$ (double)
4. $\quad$ if ($s_i == 1$) $P \leftarrow P + G$ (add)
5. }
6. Return P

Many (!) optimizations possible.

Assume the scalar and point are random.

Example – Double-and-Add

$$9\,997_{10} = 10011100001101_2$$

Naïve double-add algorithm: 13D + 6A

$$D^3 \to A \to D \to A \to D \to A \to D^5 \to A \to D \to A \to D^2 \to A$$
$$(((((2^3 + 2^0) \cdot 2^1 + 2^0) \cdot 2^1 + 2^0) \cdot 2^5 + 2^0) \cdot 2^1 + 2^0) \cdot 2^2 + 2^0 = 9\,997$$

1	10011100000
1000	10011100001
1001	100111000010
10010	100111000011
10011	10011100001100
100110	10011100001101
100111	

Example – Windowing

$$9\,997_{10} = 10011100001101_2$$

Windowing algorithm (13D + 5A)
Precompute cP with $1 \leq c < 2^w$
Assume $w = 2$, compute window: $\{P, 2P, 3P\}$ (1D + 1A)

$$(((((2 \cdot 2^2 + 1) \cdot 2^2 + 3) \cdot 2^2 + 0) \cdot 2^2 + 0) \cdot 2^2 + 3) \cdot 2^2 + 1 = 9\,997$$

10	1001110000
1000	100111000000
1001	100111000011
100100	10011100001100
100111	10011100001101
10011100	

Example – Sliding window

$$9\,997_{10} = 10011100001101_2$$

Sliding windowing algorithm (13D + 5A)
Precompute **odd** cP with $1 \leq c < 2^w$
Assume $w = 2$, compute window: $\{P, 3P\}$ (1D + 1A)

$$(((2^4 + 3) \cdot 2 + 1) \cdot 2^6 + 3) \cdot 2^2 + 1 = 9\,997$$

1	1001110000
100	100111000000
10000	100111000011
10011	10011100001100
100110	10011100001101
100111	

Example – Signed sliding window

$$9\,997_{10} = 10011100001101_2$$

Signed sliding windowing algorithm (14D + 5A)
Precompute **odd** cP with $1 \le c < 2^w$
Assume $w = 2$, compute window: $\{P, 3P\}$ (1D + 1A)
Exploit that computing negation is efficient: $-P = -(x, y) = (x, -y)$

$$(((2^2 + 1) \cdot 2^3 - 1) \cdot 2^4 + 1) \cdot 2^4 - 3 = 9\,997$$

1	1001110000
100	1001110001
101	10011100010000
101000	10011100001101
100111	

Are these approaches secure?

Double-and-Add
Windowing
Sliding windowing
Signed sliding windowing

Are these approaches secure?

	Adding $\mathcal{O}$?
Double-and-Add	✓
Windowing	✓
Sliding windowing	✗
Signed sliding windowing	✗

Are these approaches secure?

	Adding $\mathcal{O}$?	Multiple precomputed points?
Double-and-Add	✓	✗
Windowing	✓	✓
Sliding windowing	✗	✓
Signed sliding windowing	✗	✓

Are these approaches secure?

	Adding $\mathcal{O}$?	Multiple precomputed points?	Constant-time?
Double-and-Add	✓	×	×
Windowing	✓	✓	×
Sliding windowing	×	✓	×
Signed **sliding** windowing	×	✓	×

Constant-time?
Run-time is independent of the **key** and **input** to the algorithm

Implementation Attacks: Overview

Side-Channels
Power Consumption
Electromagnetic Emanation
Timing
Communication (Errors)
Heat Emanation

SCA-Attacks
Simple Power Analysis
Differential Power Analysis
Template Attacks
Timing Analysis

Non Invasive Attacks

Cryptographically Secure Algorithm
Secret
Implementation

Misuse logical implementation flaws

Logical Attacks

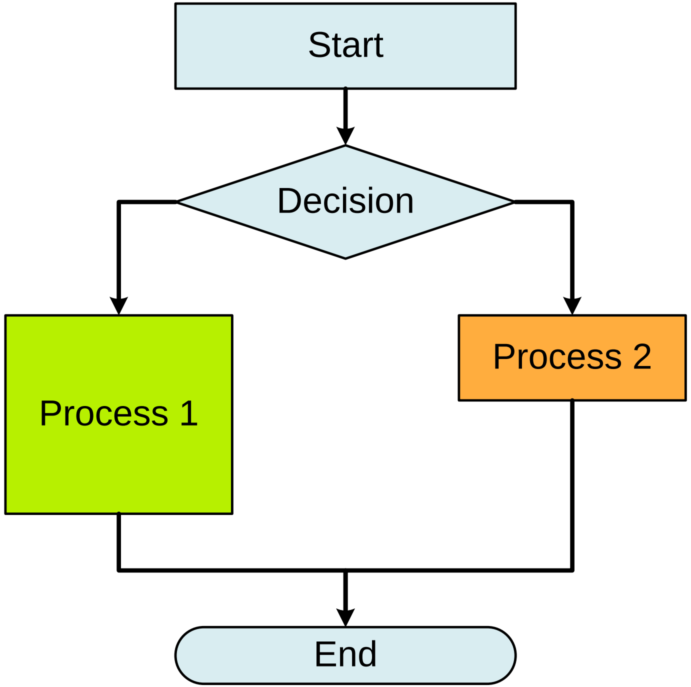
Fault Injection
Laser Beam
Power- Clock Glitches
Probes
EM Pulses
Focused Ion Beam
Environmental Stress

Fault Attacks
Attacks on Algorithm
Attacks on Program Flow
Single Bit vs. Multiple Bit
Differential Fault Attacks

Semi- / Invasive Attacks

Timing Attacks

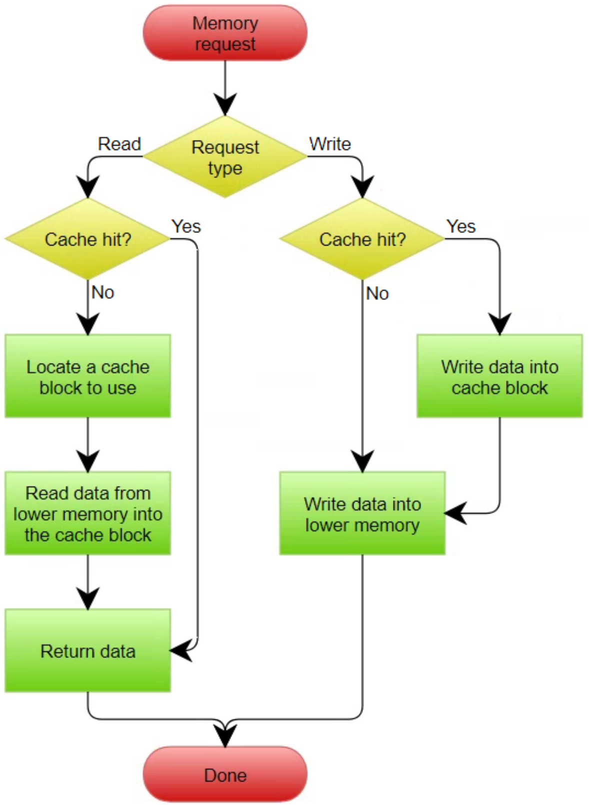
- Deduce information about the secret by measuring runtime of program
  → example of (passive) side-channel attack
  Can be performed local or remote
- Many things can influence the timing of the implementation → very hard to create truly constant-time implementations



Constant-time?
Run-time is independent of the **<u>key</u>** and **<u>input</u>** to the algorithm

Timing attacks: Cache attack

- Remote timing attacks
  (especially successful against public-key crypto)
- Local cache attacks (multi-user system)



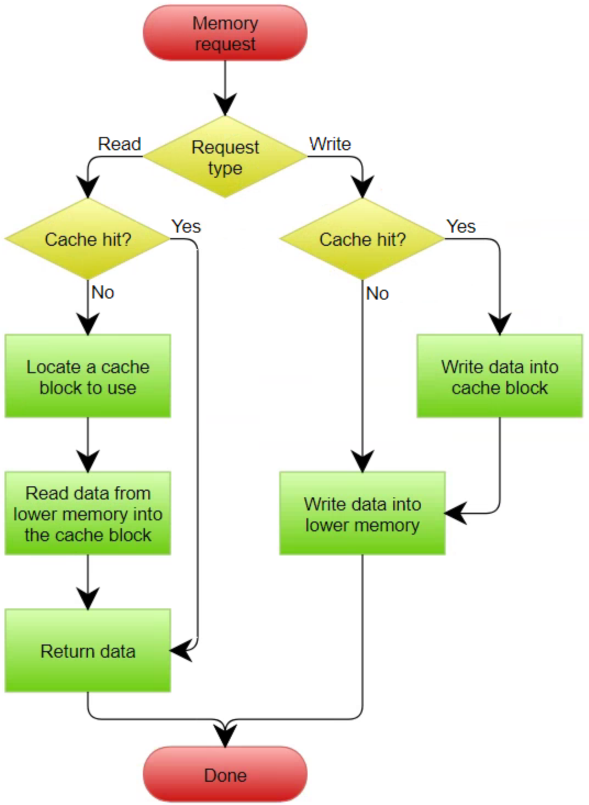
Wikipedia:
A write-through cache with no-write allocation

Constant-time?
Run-time is independent of the **key** and **input** to the algorithm

Timing attacks: Cache attack

- Remote timing attacks
  (especially successful against public-key crypto)
- Local cache attacks (multi-user system)

Example: FLUSH+RELOAD attack exploits a security weakness in the X86 architecture: monitor access to memory lines in shared pages



Wikipedia:
A write-through cache with no-write allocation

Constant-time?
Run-time is independent of the **<u>key</u>** and **<u>input</u>** to the algorithm

New developments in ECC and impact in practice

Elliptic Curve Models - Summary

Weierstrass curves

$$y^2 = x^3 + ax + b$$

- Most general form
- [+] Prime order possible
- [-] Exceptions in group law
- NIST and Brainpool curves

Elliptic Curve Models - Summary

Weierstrass curves

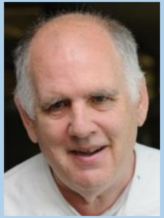
$$y^2 = x^3 + ax + b$$

- Most general form
- [+] Prime order possible
- [-] Exceptions in group law
- NIST and Brainpool curves

Montgomery curves

$$By^2 = x^3 + Ax^2 + x$$

- Subset of curves
- [-] Not prime order
- [+] Montgomery ladder



Elliptic Curve Models - Summary

Weierstrass curves

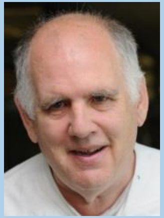
$$y^2 = x^3 + ax + b$$

- Most general form
- [+] Prime order possible
- [-] Exceptions in group law
- NIST and Brainpool curves

Montgomery curves

$$By^2 = x^3 + Ax^2 + x$$

- Subset of curves
- [-] Not prime order
- [+] Montgomery ladder



Twisted Edwards curves

$$ax^2 + y^2 = 1 + dx^2y^2$$

- Subset of curves
- [-] Not prime order
- [+] Fastest arithmetic
- [+] Some have complete group law

Elliptic Curve Models - Summary

Weierstrass curves

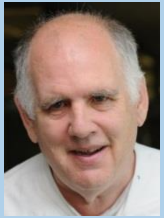
$$y^2 = x^3 + ax + b$$

- Most general form
- [+] Prime order possible
- [-] Exceptions in group law
- NIST and Brainpool curves

Montgomery curves

$$By^2 = x^3 + Ax^2 + x$$

- Subset of curves
- [-] Not prime order
- [+] Montgomery ladder



Twisted Edwards curves

$$ax^2 + y^2 = 1 + dx^2y^2$$

- Subset of curves
- [-] Not prime order
- [+] Fastest arithmetic
- [+] Some have complete group law

Montgomery ladder

- ✓ **Montgomery curves** and **Montgomery ladder** were invented to accelerate ECM.
- ✓ Regular structure
- ✓ Montgomery ladder very efficient in combination with Montgomery curve
- ✓ Small memory requirement

Algorithm 4 Montgomery ladder

Input: $\begin{cases} G \in E_{a,b}(\mathbf{F}_p) \\ n = \sum_{i=0}^{k-1} n_i 2^i, n \in \mathbf{Z}_{>0},\ 2^{k-1} \leq n < 2^k \end{cases}$

Output: $P = nG \in E_{a,b}(\mathbf{F}_p)$

1. $P \leftarrow G, Q \leftarrow G$
2. **for** $i = k - 2$ down to 0 **do**
3.   **if** $n_i = 1$ **then**
4. $(P, Q) \leftarrow (P + Q, 2Q)$
5.   **else**
6. $(P, Q) \leftarrow (2P, P + Q)$

Montgomery ladder

- ✓ **Montgomery curves** and **Montgomery ladder** were invented to accelerate ECM.
- ✓ Regular structure
- ✓ Montgomery ladder very efficient in combination with Montgomery curve
- ✓ Small memory requirement
- ✓ Can be converted in constant-time with "constant-time swapping" depending on n_i

Algorithm 4 Montgomery ladder

Input: $\begin{cases} G \in E_{a,b}(\mathbf{F}_p) \\ n = \sum_{i=0}^{k-1} n_i 2^i, n \in \mathbf{Z}_{>0},\ 2^{k-1} \leq n < 2^k \end{cases}$

Output: $P = nG \in E_{a,b}(\mathbf{F}_p)$

1. $P \leftarrow G, Q \leftarrow G$
2. **for** $i = k-2$ down to 0 **do**
3. **if** $n_i = 1$ **then**
4. $(P, Q) \leftarrow (P + Q, 2Q)$
5. **else**
6. $(P, Q) \leftarrow (2P, P + Q)$

Example: Curve25519

Cryptographic curve providing 128-bit security

Montgomery Curve

$$y^2 = x^3 + 486662x^2 + x$$

Fast ECDH →

Montgomery ladder

Curve	Double	Add
Montgomery	11	
$a = -3$ short Weierstrass	9	14

1987: Montgomery curve
2005: New ECDH speed records using Curve25519 (Montgomery curve)

Example: Curve25519

<u>Cryptographic curve providing 128-bit security</u>

Montgomery Curve

$$y^2 = x^3 + 486662x^2 + x$$

Twisted Edwards curve

$$-x^2 + y^2 = 1 - \frac{121665}{121666}x^2y^2$$

Fast ECDH → Montgomery ladder

Fast ECDSA → twisted Edwards arithmetic

Curve	Double	Add
Montgomery	11	
$a = -1$ twisted Edwards	7	8
$a = -3$ short Weierstrass	9	14

1987: Montgomery curve
2005: New ECDH speed records using Curve25519 (Montgomery curve)
2008: $a = -1$ twisted Edwards curve
2011: EdDSA → new digital signature speed records

Practice - Backwards compatibility

Implementing arithmetic on (short) Weierstrass curves makes a lot of sense.
Given a curve in another curve model one can always translate this to an equivalent Weierstrass curve

“One curve model to rule them all”

- Implement group law, counter measures etc. once.
- If new curves are proposed no need to change implementation.

Practice - Backwards compatibility

Implementing arithmetic on (short) Weierstrass curves makes a lot of sense.
Given a curve in another curve model one can always translate this to an equivalent Weierstrass curve

“One curve model to rule them all”

- Implement group law, counter measures etc. once.
- If new curves are proposed no need to change implementation.

Existing hardware / software implementations might assume

- prime order `[ almost always assumed ]`
- short Weierstrass curves `[ always assumed ]`
- with curve parameter $a = -3$ `[ not widely assumed? ]`

Practice - Backwards compatibility

Implementing arithmetic on (short) Weierstrass curves makes a lot of sense.
Given a curve in another curve model one can always translate this to an equivalent Weierstrass curve

"One curve model to rule them all"

- Implement group law, counter measures etc. once.
- If new curves are proposed no need to change implementation.

Existing hardware / software implementations might assume

- prime order `[ almost always assumed ]`
- short Weierstrass curves `[ always assumed ]`
- with curve parameter $\mathrm{a} = -3$ `[ not widely assumed? ]`

Historically this makes sense:
Standard curves $E(\mathbf{F}_p)$ with p>3 prime have these three properties
For instance see:

- NIST, FIPS 186-4, App. D: Recommended Elliptic Curves for Government Use
- SEC 2: Recommended Elliptic Curve Domain Parameters*

(* Except the three Koblitz curves secp192k1, secp224k1, secp256k1, where $a = 0$)

Practice - Backwards compatibility

Existing hardware / software implementations might assume

- prime order `[ almost always assumed ]`
  - This rules out (twisted) Edwards / Montgomery curves
  - Need additional code to avoid small-subgroup attacks

- short Weierstrass curves `[ always assumed ]`
  One curve model to rule them all: not a problem

- with curve parameter $a = -3$ `[ not widely assumed? ]`

Practice - Backwards compatibility

Existing hardware / software implementations might assume

- prime order `[ almost always assumed ]`
  - ❖ This rules out (twisted) Edwards / Montgomery curves
  - ❖ Need additional code to avoid small-subgroup attacks

- short Weierstrass curves `[ always assumed ]`
  One curve model to rule them all: not a problem

- with curve parameter $a = -3$ `[ not widely assumed? ]`

One can transform

$$y^2 = x^3 + ax + b \qquad \text{to an isomorphic} \qquad y^2 = x^3 - 3x + b'$$

if and only if there exists $u \in \mathbf{F}_p^*$ such that $u^4 = a/-3$ and $u^6 = b/b'$

Zero value / low-torsion attacks

These new curve models have an efficient complete group law.

Any disadvantages?

Zero value / low-torsion attacks

These new curve models have an efficient complete group law.

Any disadvantages?

Idea, focus on points with a zero coordinate

- zero-coordinate (Goubin's attack)
- zero-value [Akishita, Takagi]

Weierstrass	Twisted Edwards
$(x, 0)$, point of order 2	$(0,1)$, 1-torsion
$(0, \pm\sqrt{b})$	$(0, -1)$, 2-torsion
	$(\pm\sqrt{a^{-1}}, 0)$, 4-torsion

Zero value / low-torsion attacks

These new curve models have an efficient complete group law.
Any disadvantages?

Idea, focus on points with a zero coordinate

- zero-coordinate (Goubin's attack)
- zero-value [Akishita, Takagi]

Weierstrass	Twisted Edwards
$(x,0)$, point of order 2	$(0,1)$, 1-torsion
$(0,\pm\sqrt{b})$	$(0,-1)$, 2-torsion
	$(\pm\sqrt{a^{-1}},0)$, 4-torsion

- Weierstrass: $(x,0)$ does not exist when using prime order curves.
- $a=-1$ twisted Edwards: 4-torsion exists

Is this a problem for software implementations?

Zero value / low-torsion attacks

Is this a problem for software implementations?
Yes

Weierstrass	Twisted Edwards
$(x, 0)$, point of order 2	$(0,1)$, 1-torsion
$(0, \pm\sqrt{b})$	$(0, -1)$, 2-torsion
	$(\pm\sqrt{a^{-1}}, 0)$, 4-torsion

- Flush-and-reload + 4-torsion + modular reduction code
  → attack possible, torsion points make things more complicated!

- See ECC Workshop on Monday for more details
  May the Fourth Be With You: A Microarchitectural Side Channel Attack on Several Real-World Applications of Curve25519
  By Daniel Genkin

Example: EdDSA

Algorithm 1 ECDSA signature generation of a message m with the secret key d.

function ECDSA_SIGN(m, d)
  $e = \mathcal{H}(m)$
  repeat
    repeat
      Select $u \in [1, n-1]$ uniform random
      $(x, y) = uG \in E_b(\mathbb{F}_p)$
      $r = x \bmod n$
    until $r \neq 0$
    $s = u^{-1}(e + dr) \bmod n$
  until $s \neq 0$
  return (r, s)

Example: EdDSA

Algorithm 1 ECDSA signature generation of a message m with the secret key d.

function ECDSA_SIGN(m, d)
- $e = \mathcal{H}(m)$
- **repeat**
  - **repeat**
    - Select $u \in [1, n-1]$ uniform random
    - $(x, y) = uG \in E(\mathbb{F}_p)$
    - $r = x \bmod n$
  - **until** $r \neq 0$
  - $s = u^{-1}(e + dr) \bmod n$
- **until** $s \neq 0$
- **return** (r, s)

On many platforms sampling “good” random data is

- non-trivial
- insufficient entropy is available

Predictable nonce → extraction of private key

Example: EdDSA

Algorithm 1 ECDSA signature generation of a message m with the secret key d.

function ECDSA_SIGN(m, d)
  $e = \mathcal{H}(m)$
  repeat
    repeat
      Select $u \in [1, n-1]$ uniform random
      $(x, y) = uC \in E_b(\mathbb{F}_p)$
      $r = x \bmod n$
    until $r \neq 0$
    $s = u^{-1}(e + dr) \bmod n$
  until $s \neq 0$
  return (r, s)

Algorithm 2 EdDSA signature generation of a message m with the secret key k.

function EdDSA_SIGN$((m, k))$
  $m' = \mathcal{H}_1(m)$
  Retrieve or compute $(h_b, \ldots, h_{2b-1})$ from $\mathcal{H}_2(k) = (h_0, h_1, \ldots, h_{2b-1})$
  $r = \mathcal{H}_2(h_b, \ldots, h_{2b-1}, m') \bmod \ell$
  $R = rB \in E_{a,d}(\mathbb{F}_q)$
  $t = \mathcal{H}_2(\textsc{EncPoint}(R), \textsc{EncPoint}(A), m')$
  $S = (r + ts) \bmod \ell$
  return $(\textsc{EncPoint}(R), \textsc{EncInt}(S))$

- Edwards-curve Digital Signature Algorithm (EdDSA)
  - Variant of a Schnorr signature
  - Deterministic signature

Example: EdDSA

Algorithm 1 ECDSA signature generation of a message m with the secret key d.

function ECDSA_SIGN(m, d)
  $e = \mathcal{H}(m)$
  repeat
    repeat
      Select $u \in [1, n-1]$ uniform random
      $(x, y) = uG \in E_i(\mathbb{F}_p)$
      $r = x \bmod n$
    until $r \neq 0$
    $s = u^{-1}(e + dr) \bmod n$
  until $s \neq 0$
  return (r, s)

Algorithm 2 EdDSA signature generation of a message m with the secret key k.

function EdDSA_SIGN$((m, k))$
  $m' = \mathcal{H}_1(m)$
  Retrieve or compute $(h_b, \ldots, h_{2b-1})$ from $\mathcal{H}_2(k) = (h_0, h_1, \ldots, h_{2b-1})$
  $r = \mathcal{H}_2(h_b, \ldots, h_{2b-1}, m') \bmod \ell$
  $R = rB \in E_{a,d}(\mathbb{F}_q)$
  $t = \mathcal{H}_2(\text{EncPoint}(R), \text{EncPoint}(A), m')$
  $S = (r + ts) \bmod \ell$
  return $(\text{EncPoint}(R), \text{EncInt}(S))$

- Public key is point A $(= s\text{B})$
- Secret key is k, where $s = 2^n + \sum_{c \le i < n} 2^i h_i$ and $\mathcal{H}(k) = (h_0, h_1, \ldots, h_{2b-1})$

- Solves the getting “good” RNG problem, always better?

Differential Fault Analysis

The next level: moving from **passive** to **active** attacks

Fault attack

- Clock glitches
    - Temporal overclocking
- Voltage spikes
    - Temporal switch to higher (or lower) voltages
- Optical fault injection

Differential Fault Analysis

The next level: moving from **passive** to **active** attacks

Fault attack

- Clock glitches
  - Temporal overclocking
- Voltage spikes
  - Temporal switch to higher (or lower) voltages
- Optical fault injection

Controlled or uncontrolled fault

Controlled fault → inject a fault in a **target memory range**.
For instance, flipping a bit in a byte, word or any range.

Differential Fault Analysis

The next level: moving from **passive** to **active** attacks

Fault attack

- Clock glitches
  - Temporal overclocking
- Voltage spikes
  - Temporal switch to higher (or lower) voltages
- Optical fault injection

Controlled or uncontrolled fault

Controlled fault → inject a fault in a **target memory range**.
For instance, flipping a bit in a byte, word or any range.

DFA: use the difference between a faulty and a correct result to determine information about the secret key used

Example: EdDSA

- Most time-consuming operation is the elliptic curve scalar multiplication.
- Introduce a fault during the operation

→ Change the outcome of the operation

Algorithm 2 EdDSA signature generation of a message m with the secret key k.

---

function $\text{EdDSA\_sign}((m, k))$
  $m' = \mathcal{H}_1(m)$
  Retrieve or compute $(h_b, \ldots, h_{2b-1})$ from $\mathcal{H}_2(k) = (h_0, h_1, \ldots, h_{2b-1})$
  $r = \mathcal{H}_2(h_b, \ldots, h_{2b-1}, m') \bmod \ell$
  $R = rB \in E_{a,d}(\mathbb{F}_q)$
  $t = \mathcal{H}_2(\text{EncPoint}(R), \text{EncPoint}(A), m')$
  $S = (r + ts) \bmod \ell$
  return $(\text{EncPoint}(R), \text{EncInt}(S))$

Example: EdDSA

- Most time-consuming operation is the elliptic curve scalar multiplication.
- Introduce a fault during the operation

→ Change the outcome of the operation

Algorithm 2 EdDSA signature generation of a message m with the secret key k.

function EDDSA_SIGN((m, k))

$m' = \mathcal{H}_1(m)$

Retrieve or compute $(h_b, \ldots, h_{2b-1})$ from $\mathcal{H}_2(k) = (h_0, h_1, \ldots, h_{2b-1})$

$r = \mathcal{H}_2(h_b, \ldots, h_{2b-1}, m') \bmod \ell$

$R = rB \in E_{a,d}(\mathbb{F}_q)$

$t = \mathcal{H}_2(\textsc{EncPoint}(R), \textsc{EncPoint}(A), m')$

$S = (r + ts) \bmod \ell$

return $(\textsc{EncPoint}(R), \textsc{EncInt}(S))$

$$(R, S) = (rB, r + ts \bmod \ell)$$

$$(R', S') = (r'B, r + t's \bmod \ell)$$

$$t' = \mathcal{H}_2(\textsc{EncPoint}(R'), \textsc{EncPoint}(A), m')$$

Example: EdDSA

- Most time-consuming operation is the elliptic curve scalar multiplication.
- Introduce a fault during the operation

→ Change the outcome of the operation

Algorithm 2 EdDSA signature generation of a message m with the secret key k.

function EdDSA_SIGN((m, k))
  $m' = \mathcal{H}_1(m)$
  Retrieve or compute $(h_b, \ldots, h_{2b-1})$ from $\mathcal{H}_2(k) = (h_0, h_1, \ldots, h_{2b-1})$
  $r = \mathcal{H}_2(h_b, \ldots, h_{2b-1}, m') \bmod \ell$
  $R = rB \in E_{a,d}(\mathbb{F}_q)$
  $t = \mathcal{H}_2(\textsc{EncPoint}(R), \textsc{EncPoint}(A), m')$
  $S = (r + ts) \bmod \ell$
  return $(\textsc{EncPoint}(R), \textsc{EncInt}(S))$

$$(R, S) = (rB, r + ts \bmod \ell)$$

$$(R', S') = (r'B, r + t's \bmod \ell)$$

$$t' = \mathcal{H}_2(\textsc{EncPoint}(R'), \textsc{EncPoint}(A), m')$$

DFA approach

$$S - S' \equiv s(t - t') \bmod \ell$$

→ One equation with one unknown

→ compute s and check if correct using $A = sB$

Example: EdDSA

Table 1. Overview of the different proposed attacks against EdDSA which result in extracting the private key s.

where	attack	type	number of faults
Import point B	fault	uncontrolled	≥ 1
Import point A	fault	controlled	≥ 1
Hash computation of r	fault	controlled	≥ 1
Hash computation of r with fixed (unknown) output	{ fault	uncontrolled	≥ 1 }
Scalar multiplication rB	fault	uncontrolled	≥ 1
Hash computation of t	fault	controlled	≥ 1
Hash computation of t with fixed (unknown) output	{ fault	controlled	≥ 2 }
Computation of S	fault	controlled	≥ 1
Hash computation of r	DPA/DEMA	–	–

Example: Deterministic ECDSA

Table 2. Overview of the different possible attacks against deterministic ECDSA which result in extracting the private key d.

where	attack	type	number of faults
Import point G	fault	uncontrolled	≥ 1
Hash computation of u	fault	controlled	≥ 1
Hash computation of u with fixed (unknown) output	$\{$ fault	uncontrolled	$\geq 1 \}$
Scalar multiplication uG	fault	uncontrolled	≥ 1
Computation of s	fault	controlled	≥ 1
Generation of u	DPA/DEMA	–	–

Potential countermeasures

- In general: DFA countermeasures are expensive.
  - Compute twice and compare

Potential countermeasures

- In general: DFA countermeasures are expensive.
  - Compute twice and compare

- What about a hybrid approach? Use either

$$r = \mathcal{H}(h_b, \ldots, h_{2b-1}, m') \text{ or } r = \mathcal{H}(R, h_b, \ldots, h_{2b-1}, m')$$

Where R is high-quality randomness.

Potential countermeasures

- In general: DFA countermeasures are expensive.
  - Compute twice and compare
- What about a hybrid approach? Use either

$$r = \mathcal{H}(h_b, \ldots, h_{2b-1}, m') \text{ or } r = \mathcal{H}(R, h_b, \ldots, h_{2b-1}, m')$$

Where R is high-quality randomness.

Advantages

✓ Improved protection on platforms where DFA is a threat

✓ No change to

  ✓ Implementations which are not concerned with DFA

  ✓ Key generation and signature verification algorithms

However, no longer a deterministic signature scheme.

Conclusions

- Implementing elliptic curve crypto is fun,
  - Creating fast / small implementations is a nice challenge
    New developments in ECC (Curve25519) are fast but not backwards compatible.
  - Creating a "secure" implementation is **very hard**

- What does secure mean?
  - Timing attacks? Cache attacks?
  - Other passive attacks? (e.g. power)
  - Active attacks → fault injections?

- A lot of opportunity for things to go wrong in practice
  - Protocol level
  - Algorithm level
  - Implementation level

This is what makes this field so much fun!

SECURE CONNECTIONS
FOR A SMARTER WORLD