Degenerate Fault Attacks on Elliptic Curve Parameters in OpenSSL

IACR ePrint: 2019/400

Akira Takahashi[1] Mehdi Tibouchi[2]

July 2, 2019

[1]Aarhus University, Denmark

[2]NTT Secure Platform Laboratories and Kyoto University, Japan

Outline

1. Introduction

2. Theory — Singular/Supersingular Curve Point Decompression Attacks

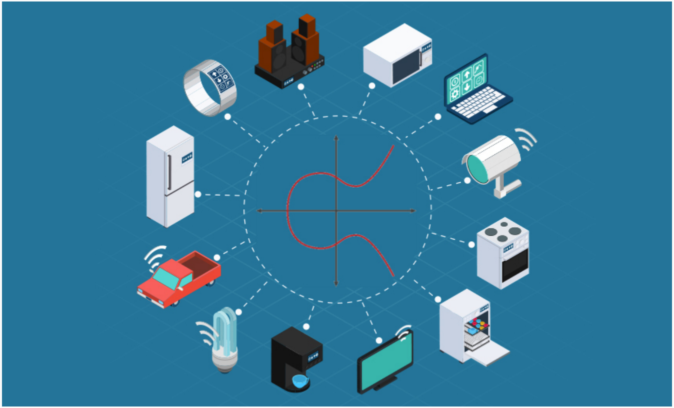
3. Practice — Attacking ECDSA and ECIES in OpenSSL

4. Beyond OpenSSL

5. Conclusion

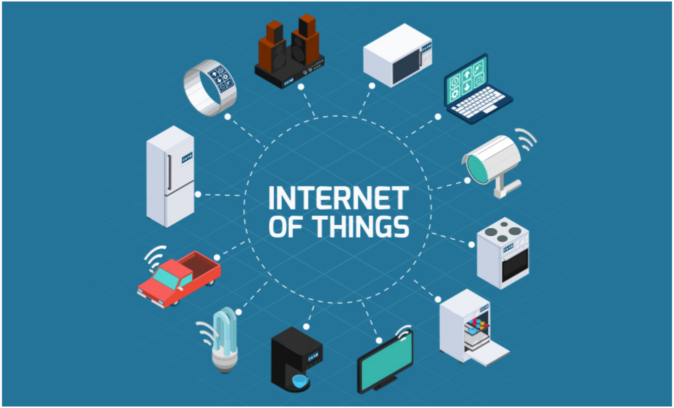
Introduction

Implementation Attacks against ECC



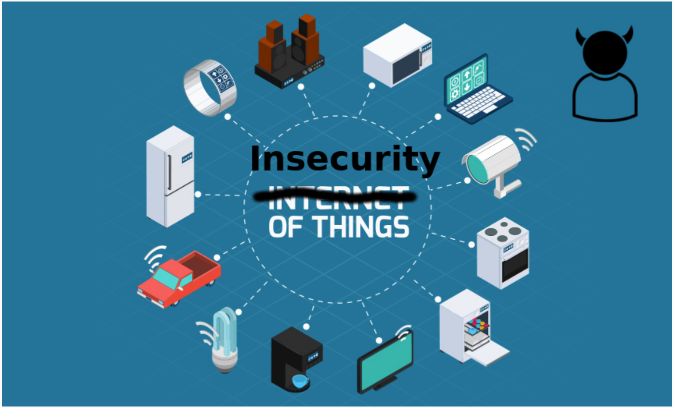
- Elliptic curve crypto is widely used in many devices

Implementation Attacks against ECC



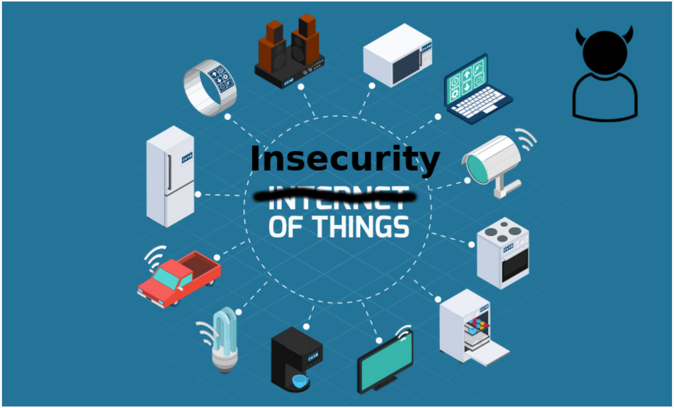
- Elliptic curve crypto is widely used in many devices
- We live in the era of IoT

Implementation Attacks against ECC



- Elliptic curve crypto is widely used in many devices
- We live in the era of IoT ≈ Insecurity of Things!

Implementation Attacks against ECC



- Elliptic curve crypto is widely used in many devices
- We live in the era of IoT $\approx$ Insecurity of Things!
- Threat of physical attacks on implementations of ECC

Invalid Curve Attacks

- Correctness attack against ECC (Antipa et al. [ABM$^+$03])

Invalid Curve Attacks

- Correctness attack against ECC (Antipa et al. [ABM+03])
- Exploits careless implementations that do not check if the input point satisfies the predefined curve equation

Invalid Curve Attacks

- Correctness attack against ECC (Antipa et al. [ABM+03])
- Exploits careless implementations that do not check if the input point satisfies the predefined curve equation
- Basic strategy of the adversary:

Invalid Curve Attacks

- Correctness attack against ECC (Antipa et al. [ABM+03])
- Exploits careless implementations that do not check if the input point satisfies the predefined curve equation
- Basic strategy of the adversary:
    1. Pick some point $\widetilde{P}$ on a weak curve $\widetilde{E}$

Invalid Curve Attacks

- Correctness attack against ECC (Antipa et al. [ABM+03])
- Exploits careless implementations that do not check if the input point satisfies the predefined curve equation
- Basic strategy of the adversary:
    1. Pick some point $\widetilde{P}$ on a weak curve $\widetilde{E}$
    2. Send $\widetilde{P}$ to the scalar multiplication algorithm

Invalid Curve Attacks

- Correctness attack against ECC (Antipa et al. [ABM+03])
- Exploits careless implementations that do not check if the input point satisfies the predefined curve equation
- Basic strategy of the adversary:
  1. Pick some point $\widetilde{P}$ on a weak curve $\widetilde{E}$
  2. Send $\widetilde{P}$ to the scalar multiplication algorithm
  3. Compute partial bits of the secret scalar k by examining an invalid output $[k]\widetilde{P}$.

Limitation of Invalid Curve Attacks

- Simple countermeasure: point validitation of the input $P = (x, y)$

$$y^2 \stackrel{?}{=} x^3 + Ax + B$$

Limitation of Invalid Curve Attacks

- Simple countermeasure: point validitation of the input $P = (x, y)$
$$y^2 \stackrel{?}{=} x^3 + Ax + B$$
- Are invalid curve attacks dead?

Limitation of Invalid Curve Attacks

- Simple countermeasure: point validitation of the input $P = (x, y)$

$$y^2 \stackrel{?}{=} x^3 + Ax + B$$

- Are invalid curve attacks dead? – NO!

Limitation of Invalid Curve Attacks

- Simple countermeasure: point validitation of the input $P = (x, y)$

$$y^2 \stackrel{?}{=} x^3 + Ax + B$$

- Are invalid curve attacks dead? – NO!
    - where there's crypto, there's a risk of fault attacks

Fault Attacks

- Active physical attacks
  - cf. SCA is passive

Fault Attacks

- Active physical attacks
  - cf. SCA is passive
- Tamper with the device to cause malfunction
  - Instruction skip
  - Memory bit-flip

Fault Attacks

- Active physical attacks
  - cf. SCA is passive
- Tamper with the device to cause malfunction
  - Instruction skip
  - Memory bit-flip
- Various methods:
  - Voltage glitch
  - Clock glitch
  - Optical attacks
  - Temperature attacks
  - Optical attacks
  - Magnetic attacks
  - etc.

Summary of the Results

We performed fault analyses on OpenSSL's elliptic curve crypto *which does the point validation*:

Summary of the Results

We performed fault analyses on OpenSSL's elliptic curve crypto *which does the point validation*:

1. Attack on ECDSA and ECIES

Summary of the Results

We performed fault analyses on OpenSSL's elliptic curve crypto *which does the point validation*:

1. Attack on ECDSA and ECIES
    - Single fault injection leads to the recovery of secret key/plaintext with almost no computational cost

Summary of the Results

We performed fault analyses on OpenSSL's elliptic curve crypto *which does the point validation*:

1. Attack on ECDSA and ECIES
    - Single fault injection leads to the recovery of secret key/plaintext with almost no computational cost
2. Attack on EC Diffie–Hellman

Summary of the Results

We performed fault analyses on OpenSSL's elliptic curve crypto *which does the point validation*:

1. Attack on ECDSA and ECIES
    - Single fault injection leads to the recovery of secret key/plaintext with almost no computational cost
2. Attack on EC Diffie–Hellman
    - Requires several faulty ciphertexts, but can recover server's secret key with practical computational cost

Summary of the Results

We performed fault analyses on OpenSSL's elliptic curve crypto *which does the point validation*:

1. Attack on ECDSA and ECIES
    - Single fault injection leads to the recovery of secret key/plaintext with almost no computational cost
2. Attack on EC Diffie–Hellman
    - Requires several faulty ciphertexts, but can recover server's secret key with practical computational cost
3. Experimentally verified that the attacks reliably work against OpenSSL installed in Raspberry Pi!

Theory — Singular/Supersingular Curve Point Decompression Attacks

SCPD Attacks Overview

- Originally described as an attack on pairing-based crypto by Blömer and Günther (FDTC'15 [BG15])

SCPD Attacks Overview

- Originally described as an attack on pairing-based crypto by Blömer and Günther (FDTC'15 [BG15])
- Variant of invalid curve attacks, making use of fault injection

SCPD Attacks Overview

- Originally described as an attack on pairing-based crypto by Blömer and Günther (FDTC’15 [BG15])
- Variant of invalid curve attacks, making use of fault injection
- We generalize & improve the SCPD attack:

SCPD Attacks Overview

- Originally described as an attack on pairing-based crypto by Blömer and Günther (FDTC'15 [BG15])
- Variant of invalid curve attacks, making use of fault injection
- We generalize & improve the SCPD attack:
    - Applicable to almost all standardized curves

SCPD Attacks Overview

- Originally described as an attack on pairing-based crypto by Blömer and Günther (FDTC'15 [BG15])
- Variant of invalid curve attacks, making use of fault injection
- We generalize & improve the SCPD attack:
    - Applicable to almost all standardized curves
    - Exploit supersingular curves for targets with non-zero j-invariant

SCPD Attacks Overview

- Originally described as an attack on pairing-based crypto by Blömer and Günther (FDTC'15 [BG15])
- Variant of invalid curve attacks, making use of fault injection
- We generalize & improve the SCPD attack:
  - Applicable to almost all standardized curves
  - Exploit supersingular curves for targets with non-zero j-invariant
  - Achievable with low-cost single fault injection

Singular Curve Point Decompression Attack

Singular Curve Point Decompression Attack

Point Compression/Decompression

- Consider a short Weierstrass form of an elliptic curve defined over $\mathbb{F}_p$:

$$E/\mathbb{F}_p : y^2 = x^3 + Ax + B$$

Point Compression/Decompression

- Consider a short Weierstrass form of an elliptic curve defined over $\mathbb{F}_p$:

$$E/\mathbb{F}_p : y^2 = x^3 + Ax + B$$

- y-coordinate is determined by x up to sign:

$$y = +\sqrt{x^3 + Ax + B} \quad \text{or} \quad -\sqrt{x^3 + Ax + B}.$$

Point Compression/Decompression

- Consider a short Weierstrass form of an elliptic curve defined over $\mathbb{F}_p$:

$$E/\mathbb{F}_p : y^2 = x^3 + Ax + B$$

- y-coordinate is determined by x up to sign:

$$y = +\sqrt{x^3 + Ax + B} \quad \text{or} \quad -\sqrt{x^3 + Ax + B}.$$

- Only the sign of y (i.e. whether y is even or odd in $\mathbb{F}_p$) needs to be stored

Point Compression/Decompression

- Consider a short Weierstrass form of an elliptic curve defined over $\mathbb{F}_p$:

$$E/\mathbb{F}_p : y^2 = x^3 + Ax + B$$

- y-coordinate is determined by x up to sign:

$$y = +\sqrt{x^3 + Ax + B} \quad \text{or} \quad -\sqrt{x^3 + Ax + B}.$$

- Only the sign of y (i.e. whether y is even or odd in $\mathbb{F}_p$) needs to be stored
- Typically used to compress public keys, but sometimes applied to base points too

Example: secp256k1 Bitcoin curve

Uncompressed base point [Sta10, §2.4.1]

```
04 79BE667E F9DCBBAC 55A06295 CE870B07
   029BFCDB 2DCE28D9 59F2815B 16F81798
   483ADA77 26A3C465 5DA4FBFC 0E1108A8
   FD17B448 A6855419 9C47D08F FB10D4B8
```

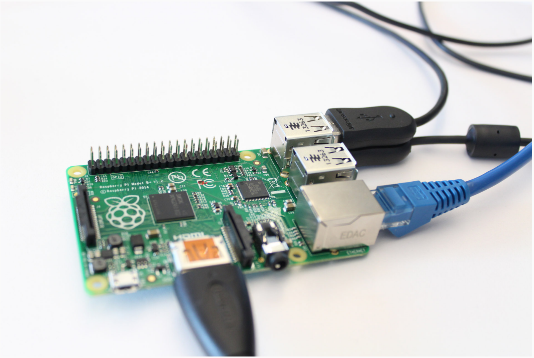
Example: secp256k1 Bitcoin curve

Compressed base point [Sta10, §2.4.1]

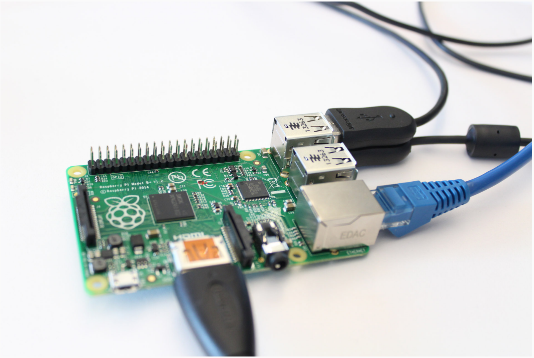
```
02 79BE667E F9DCBBAC 55A06295 CE870B07
   029BFCDB 2DCE28D9 59F2815B 16F81798
```

Singular Curve Point Decompression Attack

Attack Model

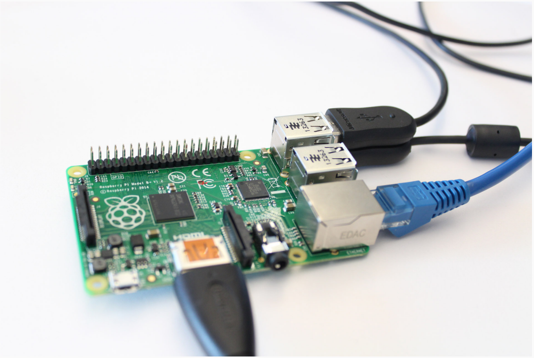


Attack Model



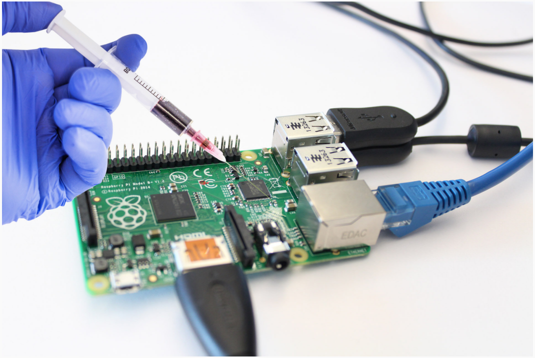
1. Compressed base point is stored in a cryptographic device

Attack Model



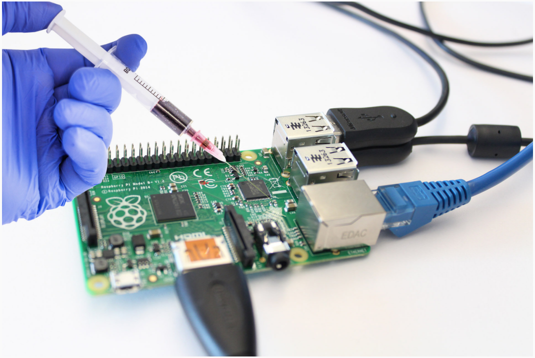
1. Compressed base point is stored in a cryptographic device
2. Base point is decompressed before passed to scalar multiplication algorithm

Attack Model



1. Compressed base point is stored in a cryptographic device
2. Base point is decompressed before passed to scalar multiplication algorithm
3. Adversary injects a fault

Attack Model



1. Compressed base point is stored in a cryptographic device
2. Base point is decompressed before passed to scalar multiplication algorithm
3. Adversary injects a fault $\rightsquigarrow$ Can skip a few instructions

Instruction Skipping Fault on Base Point Decompression (I)

Algorithm Point Decompression Algorithm

Input: $x \in \mathbb{F}_p$, $\bar{y} \in \{\texttt{0x02}, \texttt{0x03}\}$, A, B, p

Output: $P = (x, y)$: uncompressed curve point

```
y ← x^2
y ← y + A                    ▷ A = 0 for secp k and BN curves
y ← y × x
y ← y + B
y ← ±√y
Validate coordinates: y^2 =? x^3 + Ax + B
return (x, y)
```

Instruction Skipping Fault on Base Point Decompression (I)

Algorithm Point Decompression Algorithm

Input: $x \in \mathbb{F}_p$, $\bar{y} \in \{\texttt{0x02}, \texttt{0x03}\}$, A, B, p

Output: $P = (x, y)$: uncompressed curve point

$y \leftarrow x^2$
$y \leftarrow y + A$ ▷ $A = 0$ for secp k and BN curves
$y \leftarrow y \times x$
$y \leftarrow y + B$ ⚡Skip!
$y \leftarrow \pm\sqrt{y}$
Validate coordinates: $y^2 \stackrel{?}{=} x^3 + Ax + B$
return (x, y)

Instruction Skipping Fault on Base Point Decompression (I)

Algorithm Point Decompression Algorithm

Input: $x \in \mathbb{F}_p$, $\bar{y} \in \{\texttt{0x02}, \texttt{0x03}\}$, A, B, p

Output: $P = (x, y)$: uncompressed curve point

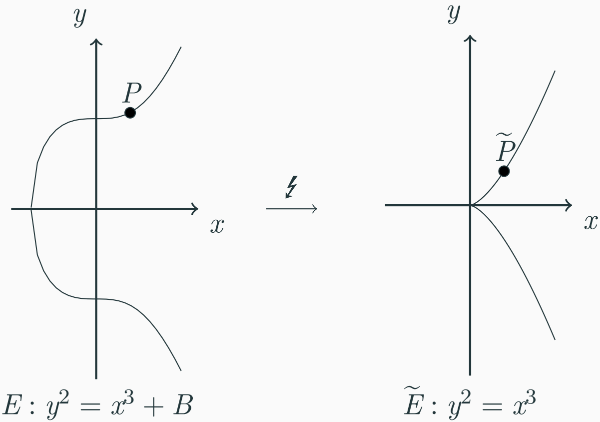
```
y ← x^2
y ← y + A                          ▷ A = 0 for secp k and BN curves
y ← y × x
y ← y + B  ⚡Skip!
y ← ±√y
Validate coordinates: y^2 =? x^3 + Ax + B  ⚡Skip!
return (x, y)
```

Instruction Skipping Fault on Base Point Decompression (II)

- y-coordinate is incorrectly reconstructed:

$$\widetilde{y}^2 = x^3 \mod p.$$

- The perturbed faulty base point $\widetilde{P} = (x, \widetilde{y})$ is reliably on singular curve $\widetilde{E}$!



Isomorphism between Singular Curve and Additive Group $\mathbb{F}_p^+$

Theorem

Let $\mathbb{F}_p^+$ be the additive group of $\mathbb{F}_p$ and $\widetilde{E}(\mathbb{F}_p)$ be the set of nonsingular $\mathbb{F}_p$-rational points on $\widetilde{E}$ including the point at infinity $O = (0 : 1 : 0)$. Then the map $\phi : \widetilde{E}(\mathbb{F}_p) \to \mathbb{F}_p^+$ with

$$(x, y) \mapsto x/y$$
$$O \mapsto 0,$$

is a group isomorphism between $\widetilde{E}(\mathbb{F}_p)$ and $\mathbb{F}_p^+$.

Isomorphism between Singular Curve and Additive Group $\mathbb{F}_p^+$

Theorem

Let $\mathbb{F}_p^+$ be the additive group of $\mathbb{F}_p$ and $\widetilde{E}(\mathbb{F}_p)$ be the set of nonsingular $\mathbb{F}_p$-rational points on $\widetilde{E}$ including the point at infinity $O = (0:1:0)$. Then the map $\phi : \widetilde{E}(\mathbb{F}_p) \to \mathbb{F}_p^+$ with

$$(x, y) \mapsto x/y$$
$$O \mapsto 0,$$

is a group isomorphism between $\widetilde{E}(\mathbb{F}_p)$ and $\mathbb{F}_p^+$.

How to Recover the Secret k

- Let $[k]\widetilde{P} = (\widetilde{x}_k, \widetilde{y}_k)$ be a faulty output

How to Recover the Secret k

- Let $[k]\widetilde{P} = (\widetilde{x}_k, \widetilde{y}_k)$ be a faulty output
- Then using the isomorphism ϕ in Theorem

$$\begin{aligned}\widetilde{x}_k/\widetilde{y}_k = \phi([k]\widetilde{P}) &= \phi(\underbrace{\widetilde{P} + \ldots + \widetilde{P}}_{k}) \\ &= \phi(\widetilde{P}) + \ldots + \phi(\widetilde{P}) \\ &= kx/\widetilde{y}.\end{aligned}$$

How to Recover the Secret k

- Let $[k]\widetilde{P} = (\widetilde{x}_k, \widetilde{y}_k)$ be a faulty output
- Then using the isomorphism ϕ in Theorem

$$\begin{aligned}\widetilde{x}_k/\widetilde{y}_k = \phi([k]\widetilde{P}) &= \phi(\underbrace{\widetilde{P} + \ldots + \widetilde{P}}_{k}) \\ &= \phi(\widetilde{P}) + \ldots + \phi(\widetilde{P}) \\ &= kx/\widetilde{y}.\end{aligned}$$

- Problem degenerates to DLP in $\mathbb{F}_p^+$ (trivial!)

How to Recover the Secret k

- Let $[k]\widetilde{P} = (\widetilde{x}_k, \widetilde{y}_k)$ be a faulty output
- Then using the isomorphism ϕ in Theorem

$$
\begin{aligned}
\widetilde{x}_k/\widetilde{y}_k = \phi([k]\widetilde{P}) &= \phi(\underbrace{\widetilde{P} + \ldots + \widetilde{P}}_{k}) \\
&= \phi(\widetilde{P}) + \ldots + \phi(\widetilde{P}) \\
&= kx/\widetilde{y}.
\end{aligned}
$$

- Problem degenerates to DLP in $\mathbb{F}_p^+$ (trivial!)
- k can be simply recovered by computing $(\widetilde{y}\widetilde{x}_k)/(x\widetilde{y}_k)$ in $\mathbb{F}_p$

What if $A \neq 0$? (New observation)

Theorem (MOV attack)

Let E' be a supersingular curve over $\mathbb{F}_p$, $p \geq 5$. Then there exists an injective, efficiently computable group homomorphism

$$e_n : E'(\mathbb{F}_p) \to \mathbb{F}_{p^2}^*$$

which can be expressed in terms of the Weil pairing on E'.

What if $A \neq 0$? (New observation)

Theorem (MOV attack)

Let E' be a supersingular curve over $\mathbb{F}_p$, $p \geq 5$. Then there exists an injective, efficiently computable group homomorphism

$$e_n : E'(\mathbb{F}_p) \to \mathbb{F}_{p^2}^*$$

which can be expressed in terms of the Weil pairing on E'.

- The curve

$$E' : y^2 = x^3 + Ax$$

has $\#E'(\mathbb{F}_p) = p + 1$ and is supersingular if $p \equiv 3 \mod 4$!

What if $A \neq 0$? (New observation)

Theorem (MOV attack)

Let E' be a supersingular curve over $\mathbb{F}_p$, $p \geq 5$. Then there exists an injective, efficiently computable group homomorphism

$$e_n : E'(\mathbb{F}_p) \to \mathbb{F}_{p^2}^*$$

which can be expressed in terms of the Weil pairing on E'.

- The curve

$$E' : y^2 = x^3 + Ax$$

  has $\#E'(\mathbb{F}_p) = p + 1$ and is supersingular if $p \equiv 3 \mod 4$!
- We can apply Menezes–Okamoto–Vanstone (MOV) attack!

What if $A \neq 0$? (New observation)

Theorem (MOV attack)

Let E' be a supersingular curve over $\mathbb{F}_p$, $p \geq 5$. Then there exists an injective, efficiently computable group homomorphism

$$e_n : E'(\mathbb{F}_p) \to \mathbb{F}_{p^2}^*$$

which can be expressed in terms of the Weil pairing on E'.

- The curve

$$E' : y^2 = x^3 + Ax$$

  has $\#E'(\mathbb{F}_p) = p + 1$ and is supersingular if $p \equiv 3 \mod 4$!
- We can apply Menezes–Okamoto–Vanstone (MOV) attack!
- The DLP on E' is no harder than the DLP in the multiplicative group $\mathbb{F}_{p^2}^*$.

What if $A \neq 0$? (New observation)

Theorem (MOV attack)

Let E' be a supersingular curve over $\mathbb{F}_p$, $p \geq 5$. Then there exists an injective, efficiently computable group homomorphism

$$e_n : E'(\mathbb{F}_p) \to \mathbb{F}_{p^2}^*$$

which can be expressed in terms of the Weil pairing on E'.

- The curve

  $$E' : y^2 = x^3 + Ax$$

  has $\#E'(\mathbb{F}_p) = p + 1$ and is supersingular if $p \equiv 3 \mod 4$!
- We can apply Menezes–Okamoto–Vanstone (MOV) attack!
- The DLP on E' is no harder than the DLP in the multiplicative group $\mathbb{F}_{p^2}^*$.
- Tractable for most standardized parameters

Practicality Issues

- Requires a double fault to skip the point validation
    - Hard to realize
    - Especially on larger embedded platforms with high frequency chips and modern OSes

Practicality Issues

- Requires a double fault to skip the point validation
  - Hard to realize
  - Especially on larger embedded platforms with high frequency chips and modern OSes
- Previous work targeted an AVR microcontroller running the pairing-based BLS signature

Practicality Issues

- Requires a double fault to skip the point validation
  - Hard to realize
  - Especially on larger embedded platforms with high frequency chips and modern OSes
- Previous work targeted an AVR microcontroller running the pairing-based BLS signature
  - Not so widely used setting

Practicality Issues

- Requires a double fault to skip the point validation
  - Hard to realize
  - Especially on larger embedded platforms with high frequency chips and modern OSes
- Previous work targeted an AVR microcontroller running the pairing-based BLS signature
  - Not so widely used setting

Can SCPD attacks be more practical?

Practicality Issues

- Requires a double fault to skip the point validation
  - Hard to realize
  - Especially on larger embedded platforms with high frequency chips and modern OSes
- Previous work targeted an AVR microcontroller running the pairing-based BLS signature
  - Not so widely used setting

Can SCPD attacks be more practical?

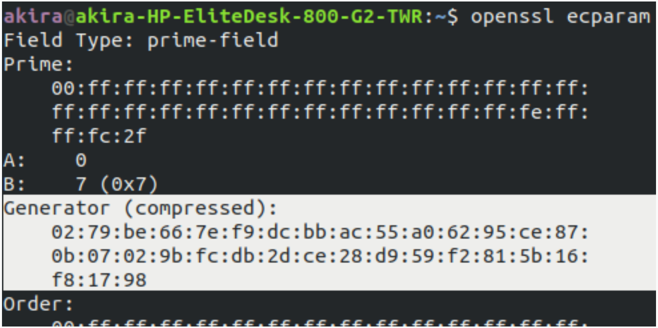
Practice — Attacking ECDSA and ECIES in OpenSSL

- OpenSSL's `ecparam` command allows users to generate EC key files with:

```
akira@akira-HP-EliteDesk-800-G2-TWR:~$ openssl ecparam
Field Type: prime-field
Prime:
    00:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:
    ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:fe:ff:
    ff:fc:2f
A:    0
B:    7 (0x7)
Generator (compressed):
    02:79:be:66:7e:f9:dc:bb:ac:55:a0:62:95:ce:87:
    0b:07:02:9b:fc:db:2d:ce:28:d9:59:f2:81:5b:16:
    f8:17:98
Order:
```

OpenSSL EC Key Files

- OpenSSL's `ecparam` command allows users to generate EC key files with:
    - Explicit curve parameters (`-param_enc explicit`)



OpenSSL EC Key Files

- OpenSSL's `ecparam` command allows users to generate EC key files with:
  - Explicit curve parameters (`-param_enc explicit`)
  - Compressed base point (`-conv_form compressed`)

```
akira@akira-HP-EliteDesk-800-G2-TWR:~$ openssl ecparam
Field Type: prime-field
Prime:
    00:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:
    ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:fe:ff:
    ff:fc:2f
A:    0
B:    7 (0x7)
Generator (compressed):
    02:79:be:66:7e:f9:dc:bb:ac:55:a0:62:95:ce:87:
    0b:07:02:9b:fc:db:2d:ce:28:d9:59:f2:81:5b:16:
    f8:17:98
Order:
```

OpenSSL EC Key Files

- OpenSSL's `ecparam` command allows users to generate EC key files with:
  - Explicit curve parameters (`-param_enc explicit`)
  - Compressed base point (`-conv_form compressed`)
  - Compressed public key (`-conv_form compressed`)

```
akira@akira-HP-EliteDesk-800-G2-TWR:~$ openssl ecparam
Field Type: prime-field
Prime:
    00:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:
    ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:ff:fe:ff:
    ff:fc:2f
A:    0
B:    7 (0x7)
Generator (compressed):
    02:79:be:66:7e:f9:dc:bb:ac:55:a0:62:95:ce:87:
    0b:07:02:9b:fc:db:2d:ce:28:d9:59:f2:81:5b:16:
    f8:17:98
Order:
```

How to Attack with a Single Fault

Input: Domain parameters in raw binary formats

Output: Domain parameters in `BIGNUM` type

$p \leftarrow$ `BN_bin2bn`(p_{bin})

$A \leftarrow$ `BN_bin2bn`(A_{bin})

$B \leftarrow$ `BN_bin2bn`(B_{bin})

$x \leftarrow$ `BN_bin2bn`(x_{bin})

$P \leftarrow$ `Decomp`$(\bar{P} = (x, \bar{y}), p, A, B)$

Validate $y^2 \stackrel{?}{=} x^3 + Ax + B$

return (p, A, B, P)

- `BIGNUM`: OpenSSL's data structure representing a multiprecision integer
- `BN_bin2bn()`: utility function which converts a raw byte array to a `BIGNUM` object

How to Attack with a Single Fault

Input: Domain parameters in raw binary formats

Output: Domain parameters in `BIGNUM` type

1: $p \leftarrow$ `BN_bin2bn`(p_{bin})

2: $A \leftarrow$ `BN_bin2bn`(A_{bin})

3: $B \leftarrow$ `BN_bin2bn`(B_{bin})

4: $x \leftarrow$ `BN_bin2bn`(x_{bin})

5: $\widetilde{P} \leftarrow$ `Decomp`$(\bar{P} = (x, \bar{y}), p, A, B)$ ⚡SCPD fault

6: Validate $y^2 \stackrel{?}{=} x^3 + Ax + B$ ⚡SCPD fault

7: **return** $(p, A, B, \widetilde{P})$

- `BIGNUM`: OpenSSL's data structure representing a multiprecision integer
- `BN_bin2bn()`: utility function which converts a raw byte array to a `BIGNUM` object

Input: Domain parameters in raw binary formats

Output: Domain parameters in `BIGNUM` type

1: $p \leftarrow$ `BN_bin2bn`(p_{bin})

2: $A \leftarrow$ `BN_bin2bn`(A_{bin})

3: $0 \leftarrow$ `BN_bin2bn`(B_{bin}) ⚡Our fault

4: $x \leftarrow$ `BN_bin2bn`(x_{bin})

5: $\widetilde{P} \leftarrow$ `Decomp`$(\bar{P} = (x, \bar{y}), p, A, 0)$

6: Validate $y^2 \stackrel{?}{=} x^3 + Ax + 0$

7: **return** $(p, A, 0, \widetilde{P})$

- `BIGNUM`: OpenSSL's data structure representing a multiprecision integer
- `BN_bin2bn()`: utility function which converts a raw byte array to a `BIGNUM` object

Realization of Our Attack Model

- Actual fault attack targets a certain CPU instruction

Figure 10: Complete assembly code for `BN_bin2bn()` function, generated by GCC 6.3.0 in Raspberry Pi

```
        .arch armv6
        .align      2
        .global     BN_bin2bn
        .syntax unified
        .arm
        .fpu vfp
        .type       BN_bin2bn, %function
BN_bin2bn:
        @ args = 0, pretend = 0, frame = 0
        @ frame_needed = 0, uses_anonymous_args = 0
        push        {r4, r5, r6, r7, r8, r9, r10, lr}
        subs        r8, r2, #0
        mov         r4, r0
        mov         r6, r1
        movne       r10, #0
        beq         .L351
.L330:

        cmp         r6, #0
        ble         .L332
        ldrb    r3, [r4]
        cmp         r3, #0
        bne         .L332
        add         r3, r4, #1
.L334:
        subs        r6, r6, #1
        mov         r4, r3
        beq         .L333
        ldrb        r2, [r3]
        add         r3, r3, #1          @ SKIP!!
        cmp         r2, #0
        beq         .L334
.L332:
        cmp         r6, #0
        bne         .L335           @ SKIP!!
.L333:
        mov         r9, r8
        mov         r3, #0
        str         r3, [r8, #4]
.L329:
        mov         r0, r9
        pop         {r4, r5, r6, r7, r8, r9, r10, pc}
.L335:
        sub         r5, r6, #1
        mov         r0, r8
        lsr         r7, r5, #2
        add         r7, r7, #1
        mov         r1, r7
        bl        bn_wexpand(PLT)
        and         r5, r5, #3
        subs        r9, r0, #0
        beq         .L352
        mov         r2, #0
        mov         r3, r2
        add         r6, r4, r6
        mov         r0, r2
        str         r7, [r8, #4]
        str         r2, [r8, #12]
.L337:
        ldrb        r1, [r4], #1
        cmp         r5, #0
        sub         r5, r5, #1
        orr         r3, r1, r3, lsl #8
        beq         .L338                   @ SKIP!!
        cmp         r4, r6
        bne         .L337
        mov         r0, r8
        bl        bn_correct_top(PLT)
        mov         r9, r8
.L353:
        mov         r0, r9
        pop         {r4, r5, r6, r7, r8, r9, r10, pc}
.L338:
        ldr         r2, [r8]
        sub         r7, r7, #1
        cmp         r4, r6
        str         r3, [r2, r7, lsl #2] @ SKIP!!
        mov         r5, #3
        mov         r3, r0
        bne         .L337
        mov         r0, r8
        bl        bn_correct_top(PLT)
        mov         r9, r8
        b         .L353
.L351:
        bl        BN_new(PLT)
        subs        r8, r0, #0
        movne       r10, r8
        bne         .L330
        mov         r9, r8
        b         .L329
.L352:
        mov         r0, r10
        bl        BN_free(PLT)
        b         .L329
        .size       BN_bin2bn, .-BN_bin2bn
```

Realization of Our Attack Model

- Actual fault attack targets a certain CPU instruction
- We identified 4 possibly vulnerable instructions in `BN_bin2bn()`'s assembly code when compiled in Rasperry Pi

Figure 10: Complete assembly code for `BN_bin2bn()` function, generated by GCC 6.3.0 in Raspberry Pi

```
        .arch armv6
        .align      2
        .global     BN_bin2bn
        .syntax unified
        .arm
        .fpu vfp
        .type       BN_bin2bn, %function
BN_bin2bn:
        @ args = 0, pretend = 0, frame = 0
        @ frame_needed = 0, uses_anonymous_args = 0
        push        {r4, r5, r6, r7, r8, r9, r10, lr}
        subs        r8, r2, #0
        mov         r4, r0
        mov         r6, r1
        movne       r10, #0
        beq         .L351
.L330:

        cmp         r6, #0
        ble         .L332
        ldrb    r3, [r4]
        cmp         r3, #0
        bne         .L332
        add         r3, r4, #1
.L334:
        subs        r6, r6, #1
        mov         r4, r3
        beq         .L333
        ldrb        r2, [r3]
        add         r3, r3, #1          @ SKIP!!
        cmp         r2, #0
        beq         .L334
.L332:
        cmp         r6, #0
        bne         .L335           @ SKIP!!
.L333:
        mov         r9, r8
        mov         r3, #0
        str         r3, [r8, #4]
.L329:
        mov         r0, r9
        pop         {r4, r5, r6, r7, r8, r9, r10, pc}
.L335:
        sub         r5, r6, #1
        mov         r0, r8
        lsr         r7, r5, #2
        add         r7, r7, #1
        mov         r1, r7
        bl          bn_wexpand(PLT)
        and         r5, r5, #3
        subs        r9, r0, #0
        beq         .L352
        mov         r2, #0
        mov         r3, r2
        add         r6, r4, r6
        mov         r0, r2
        str         r7, [r8, #4]
        str         r2, [r8, #12]
.L337:
        ldrb        r1, [r4], #1
        cmp         r5, #0
        sub         r5, r5, #1
        orr         r3, r1, r3, lsl #8
        beq         .L338                   @ SKIP!!
        cmp         r4, r6
        bne         .L337
        mov         r0, r8
        bl          bn_correct_top(PLT)
        mov         r9, r8
.L353:
        mov         r0, r9
        pop         {r4, r5, r6, r7, r8, r9, r10, pc}
.L338:
        ldr         r2, [r8]
        sub         r7, r7, #1
        cmp         r4, r6
        str         r3, [r2, r7, lsl #2] @ SKIP!!
        mov         r5, #3
        mov         r3, r0
        bne         .L337
        mov         r0, r8
        bl          bn_correct_top(PLT)
        mov         r9, r8
        b           .L353
.L351:
        bl          BN_new(PLT)
        subs        r8, r0, #0
        movne       r10, r8
        bne         .L330
        mov         r9, r8
        b           .L329
.L352:
        mov         r0, r10
        bl          BN_free(PLT)
        b           .L329
        .size       BN_bin2bn, .-BN_bin2bn
```

Realization of Our Attack Model

- Actual fault attack targets a certain CPU instruction
- We identified 4 possibly vulnerable instructions in `BN_bin2bn()`'s assembly code when compiled in Rasperry Pi
- Quick experiment: comment out each target line ⇝ the function returned 0!

Figure 10: Complete assembly code for `BN_bin2bn()` function, generated by GCC 6.3.0 in Raspberry Pi

```
        .arch armv6
        .align          2
        .global         BN_bin2bn
        .syntax unified
        .arm
        .fpu vfp
        .type           BN_bin2bn, %function
BN_bin2bn:
        @ args = 0, pretend = 0, frame = 0
        @ frame_needed = 0, uses_anonymous_args = 0
        push            {r4, r5, r6, r7, r8, r9, r10, lr}
        subs            r8, r2, #0
        mov             r4, r0
        mov             r6, r1
        movne           r10, #0
        beq             .L351
.L330:

        cmp             r6, #0
        ble             .L332
        ldrb    r3, [r4]
        cmp             r3, #0
        bne             .L332
        add             r3, r4, #1
.L334:
        subs            r6, r6, #1
        mov             r4, r3
        beq             .L333
        ldrb            r2, [r3]
        add             r3, r3, #1          @ SKIP!!
        cmp             r2, #0
        beq             .L334
.L332:
        cmp             r6, #0
        bne             .L335          @ SKIP!!
.L333:
        mov             r9, r8
        mov             r3, #0
        str             r3, [r8, #4]
.L329:
        mov             r0, r9
        pop             {r4, r5, r6, r7, r8, r9, r10, pc}
.L335:
        sub             r5, r6, #1
        mov             r0, r8
        lsr             r7, r5, #2
        add             r7, r7, #1
        mov             r1, r7
        bl              bn_wexpand(PLT)
        and             r5, r5, #3
        subs            r9, r0, #0
        beq             .L352
        mov             r2, #0
        mov             r3, r2
        add             r6, r4, r6
        mov             r0, r2
        str             r7, [r8, #4]
        str             r2, [r8, #12]
.L337:
        ldrb            r1, [r4], #1
        cmp             r5, #0
        sub             r5, r5, #1
        orr             r3, r1, r3, lsl #8
        beq             .L338                  @ SKIP!!
        cmp             r4, r6
        bne             .L337
        mov             r0, r8
        bl              bn_correct_top(PLT)
        mov             r9, r8
.L353:
        mov             r0, r9
        pop             {r4, r5, r6, r7, r8, r9, r10, pc}
.L338:
        ldr             r2, [r8]
        sub             r7, r7, #1
        cmp             r4, r6
        str             r3, [r2, r7, lsl #2] @ SKIP!!
        mov             r5, #3
        mov             r3, r0
        bne             .L337
        mov             r0, r8
        bl              bn_correct_top(PLT)
        mov             r9, r8
        b               .L353
.L351:
        bl              BN_new(PLT)
        subs            r8, r0, #0
        movne           r10, r8
        bne             .L330
        mov             r9, r8
        b               .L329
.L352:
        mov             r0, r10
        bl              BN_free(PLT)
        b               .L329
        .size           BN_bin2bn, .-BN_bin2bn
```

Algorithm ECDSA signature generation [JMV01]

Input: P: base point of prime order n, $d \in \mathbb{Z}/n\mathbb{Z}$: secret key, $Q = [d]P$: public key, $M \in \{0,1\}^*$: message to be signed

Output: a valid signature (r, s)

1: $k \xleftarrow{\$} \mathbb{Z}/n\mathbb{Z}$
2: $(x_k, y_k) \leftarrow [k]P$
3: $r \leftarrow x_k \mod n$
4: $h \leftarrow H(M)$
5: $s \leftarrow k^{-1}(h + rd) \mod n$
6: **return** (r, s)

Effect on ECDSA

Algorithm ECDSA signature generation [JMV01]

Input: P: base point of prime order n, $d \in \mathbb{Z}/n\mathbb{Z}$: secret key, $Q = [d]P$: public key, $M \in \{0,1\}^*$: message to be signed

Output: a valid signature (r, s)

$k \leftarrow_{\$} \mathbb{Z}/n\mathbb{Z}$

$(\widetilde{x}_k, \widetilde{y}_k) \leftarrow [k]\widetilde{P}$

$\widetilde{r} \leftarrow \widetilde{x}_k \mod n$

$h \leftarrow H(M)$

$\widetilde{s} \leftarrow k^{-1}(h + \widetilde{r}d) \mod n$

return $(\widetilde{r}, \widetilde{s})$

Effect on ECDSA

Algorithm ECDSA signature generation [JMV01]

Input: P: base point of prime order n, $d \in \mathbb{Z}/n\mathbb{Z}$: secret key, $Q = [d]P$: public key, $M \in \{0,1\}^*$: message to be signed

Output: a valid signature (r, s)

1: $k \leftarrow_{\$} \mathbb{Z}/n\mathbb{Z}$
2: $(\widetilde{x}_k, \widetilde{y}_k) \leftarrow [k]\widetilde{P}$
3: $\widetilde{r} \leftarrow \widetilde{x}_k \mod n$
4: $h \leftarrow H(M)$
5: $\widetilde{s} \leftarrow k^{-1}(h + \widetilde{r}d) \mod n$
6: **return** $(\widetilde{r}, \widetilde{s})$

Once k is obtained, the secret key d is directly exposed:

$$d = (\widetilde{s}k - h)/\widetilde{r} \mod n$$

Effect on SM2-ECIES (for OpenSSL ver. $\geq 1.1.1$)

Algorithm SM2-ECIES encryption [SL14]

Input: $Q \in E(\mathbb{F}_p)$: public key, $M \in \{0,1\}^*$: plaintext

Output: ciphertext (C_1, C_2, C_3)

1: $k \leftarrow_{\$} \mathbb{Z}/n\mathbb{Z}$

2: $C_1 = (x_k, y_k) \leftarrow [k]P$

3: $(x', y') \leftarrow [k]Q$

4: $K \leftarrow \mathsf{KDF}(x'||y', |M|)$

5: $C_2 \leftarrow M \oplus K$

6: $C_3 \leftarrow H(x'||y'||M)$

7: **return** (C_1, C_2, C_3)

Effect on SM2-ECIES (for OpenSSL ver. $\geq 1.1.1$)

Algorithm SM2-ECIES encryption [SL14]

Input: $Q \in E(\mathbb{F}_p)$: public key, $M \in \{0,1\}^*$: plaintext

Output: ciphertext (C_1, C_2, C_3)

1: $k \leftarrow_{\$} \mathbb{Z}/n\mathbb{Z}$

2: $C_1 = (\widetilde{x}_k, \widetilde{y}_k) \leftarrow [k]\widetilde{P}$

3: $(x', y') \leftarrow [k]\widetilde{Q}$

4: $K \leftarrow \mathsf{KDF}(x' || y', |M|)$

5: $C_2 \leftarrow M \oplus K$

6: $C_3 \leftarrow H(x' || y' || M)$

7: **return** (C_1, C_2, C_3)

Effect on SM2-ECIES (for OpenSSL ver. $\geq 1.1.1$)

Algorithm SM2-ECIES encryption [SL14]

Input: $Q \in E(\mathbb{F}_p)$: public key, $M \in \{0,1\}^*$: plaintext

Output: ciphertext (C_1, C_2, C_3)

1: $k \leftarrow_{\$} \mathbb{Z}/n\mathbb{Z}$
2: $C_1 = (\widetilde{x}_k, \widetilde{y}_k) \leftarrow [k]\widetilde{P}$
3: $(x', y') \leftarrow [k]\widetilde{Q}$
4: $K \leftarrow \mathsf{KDF}(x' || y', |M|)$
5: $C_2 \leftarrow M \oplus K$
6: $C_3 \leftarrow H(x' || y' || M)$
7: **return** (C_1, C_2, C_3)

- Once K is obtained, the plaintext can be recovered:

$$M = C_2 \oplus K.$$

Practical Experiment

- Target:
  - Raspberry Pi Model B
  - OpenSSL 1.1.1: latest release as of November 2018
  - ECDSA/SM2-ECIES over secp256k1

Practical Experiment

- Target:
  - Raspberry Pi Model B
  - OpenSSL 1.1.1: latest release as of November 2018
  - ECDSA/SM2-ECIES over secp256k1
- ChipWhisperer-Lite side-channel/fault analysis evaluation board

Experimental Setup (I)

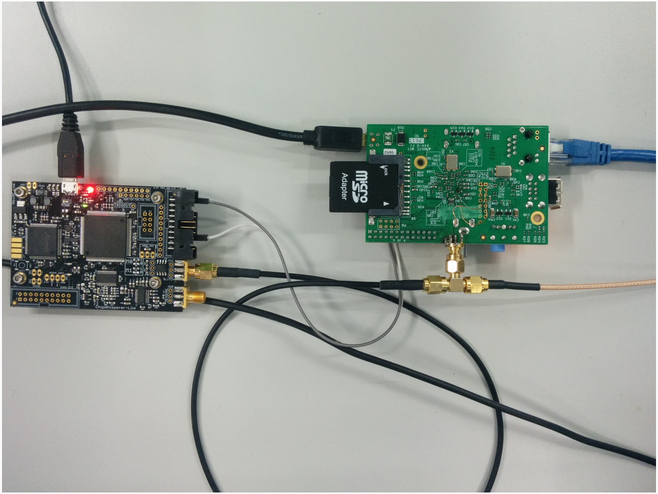


Figure 1: ChipWhisperer-Lite evaluation board connected to Raspberry Pi Model B

Experimental Setup (II)

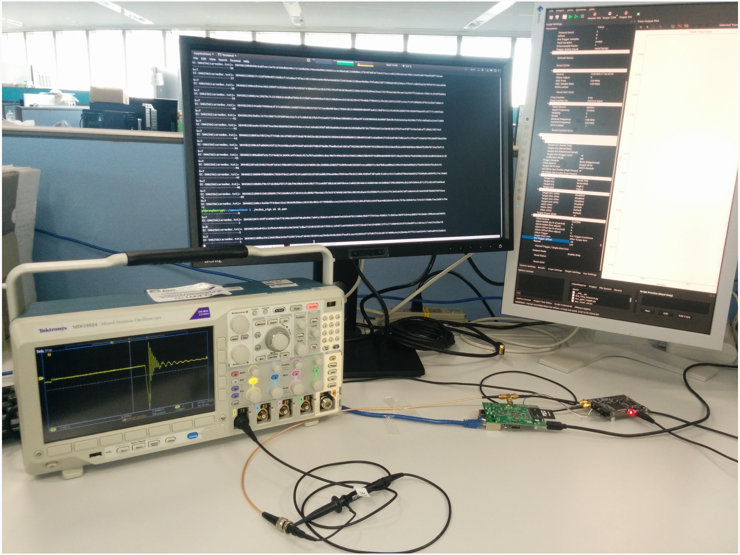
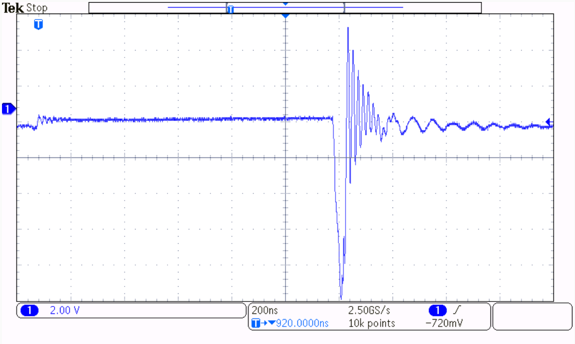


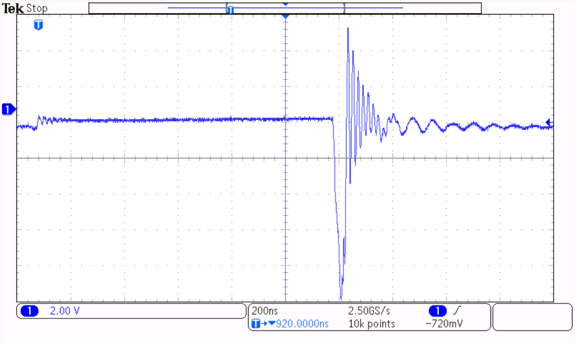
Figure 2: Overview of the experimental setup

Experimental Setup (III)



- Inserted a single voltage glitch

Experimental Setup (III)



- Inserted a single voltage glitch
- Found the suitable parameters causing reliably reproducible misbehavior of Raspberry Pi:
  - Enable-only glitches repeated 127 times
  - Offset 10 clock cycles

Experimental Result

Success	No effect	Program crash	OS crash	Total
95	813	89	3	1000

- $\approx 10\%$ success rate
- Still serious enough since the adversary requires only one successful instance to recover the secret

Beyond OpenSSL

Bitcoin Wallets

- secp256k1 curve is nowadays a high-profile target owing to its use in Bitcoin protocol

Bitcoin Wallets

- secp256k1 curve is nowadays a high-profile target owing to its use in Bitcoin protocol
- We investigated several major open-source bitcoin wallet implementations

Bitcoin Wallets

- secp256k1 curve is nowadays a high-profile target owing to its use in Bitcoin protocol
- We investigated several major open-source bitcoin wallet implementations
- Turned out they do not use decompression technique for base points:

Bitcoin Wallets

- secp256k1 curve is nowadays a high-profile target owing to its use in Bitcoin protocol
- We investigated several major open-source bitcoin wallet implementations
- Turned out they do not use decompression technique for base points:
  - ✓ libsecp256k1

Bitcoin Wallets

- secp256k1 curve is nowadays a high-profile target owing to its use in Bitcoin protocol
- We investigated several major open-source bitcoin wallet implementations
- Turned out they do not use decompression technique for base points:
  - ✓ libsecp256k1
  - ✓ Trezor

Bitcoin Wallets

- secp256k1 curve is nowadays a high-profile target owing to its use in Bitcoin protocol
- We investigated several major open-source bitcoin wallet implementations
- Turned out they do not use decompression technique for base points:
  - ✓ libsecp256k1
  - ✓ Trezor
  - ✓ Ledger

Bitcoin Wallets

- secp256k1 curve is nowadays a high-profile target owing to its use in Bitcoin protocol
- We investigated several major open-source bitcoin wallet implementations
- Turned out they do not use decompression technique for base points:
  - ✓ libsecp256k1
  - ✓ Trezor
  - ✓ Ledger
- More exhaustive evaluation will be required!
  - Some PoC implementation does use the compressed BP

Conclusion

Conclusion

- Brought the invalid curve attacks closer to practice with the help of low-cost single fault injection

Conclusion

- Brought the invalid curve attacks closer to practice with the help of low-cost single fault injection
- Demonstrated the attacks in a practical scenario
    - OpenSSL installed in Raspberry Pi

Conclusion

- Brought the invalid curve attacks closer to practice with the help of low-cost single fault injection
- Demonstrated the attacks in a practical scenario
    - OpenSSL installed in Raspberry Pi
- Lesson: Never apply point compression/decompression to base points!

Countermeasure & Future Work

- Suggestion

Countermeasure & Future Work

- Suggestion
  - OpenSSL command line should deprecate `-conv_form compressed` option

Countermeasure & Future Work

- Suggestion
  - OpenSSL command line should deprecate `-conv_form compressed` option
  - Notified to OpenSSL management committee

Countermeasure & Future Work

- Suggestion
  - OpenSSL command line should deprecate `-conv_form compressed` option
  - Notified to OpenSSL management committee
- Future work

Countermeasure & Future Work

- Suggestion
  - OpenSSL command line should deprecate `-conv_form compressed` option
  - Notified to OpenSSL management committee
- Future work
  - Fault without physical access to the target?
    - Rowhammer.js

Countermeasure & Future Work

- Suggestion
  - OpenSSL command line should deprecate `-conv_form compressed` option
  - Notified to OpenSSL management committee
- Future work
  - Fault without physical access to the target?
    - Rowhammer.js
  - Investigate more cryptocurrency wallets/libraries

Tack!

`https://ia.cr/2019/400`

Questions?

AARHUS UNIVERSITET

NTT

References i

- Adrian Antipa, Daniel R. L. Brown, Alfred Menezes, René Struik, and Scott A. Vanstone.
  Validation of Elliptic Curve Public Keys.
  In *PKC 2003*, volume 2567 of *LNCS*, pages 211–223. Springer, 2003.
- Johannes Blömer and Peter Günther.
  Singular Curve Point Decompression Attack.
  In *FDTC 2015*, pages 71–84. IEEE, 2015.
- Freepik.
  Icons made by Freepik from Flaticon.com is licensed by CC 3.0 BY.
  `http://www.flaticon.com`.

- Don Johnson, Alfred Menezes, and Scott A. Vanstone.
  The Elliptic Curve Digital Signature Algorithm (ECDSA).
  International Journal of Information Security, 1(1):36–63, 2001.
- Sean Shen and XiaoDong Lee.
  SM2 Digital Signature Algorithm.
  IETF, 2014.
  draft-shen-sm2-ecdsa-02.
- Standards for Efficient Cryptography Group (SECG).
  SEC 2: Recommended Elliptic Curve Domain Parameters, 2010.
  Version 2.0.