Exploiting dummy codes in Elliptic Curve Cryptography

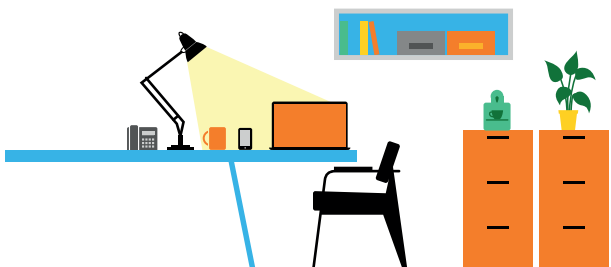
Andy Russon

4 June 2020

orange™

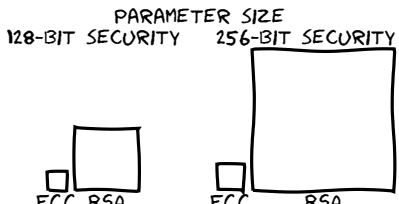
About me

- PhD thesis on elliptic curves
- Orange, and Université de Rennes 1
- Risk assessment and audit
- Interest: challenges (root-me, CryptoHack), korean movies, science-fiction



About Elliptic Curve Cryptography:

- Public-key cryptography with small parameters, keys, signatures, etc

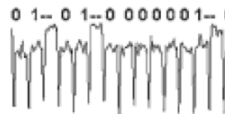


- Protocols: TLS 1.3, SSH, Bitcoin, Signal, etc
- Servers, smart cards, IoT devices, etc

Introduction

Hard to implement secure and efficient cryptography.

- Depends on threat model (physical access to the device, etc)
- One protection can lead to a vulnerability
- Passive attacks: timing, power analysis, etc



- Active attacks: differential fault analysis, C safe-errors

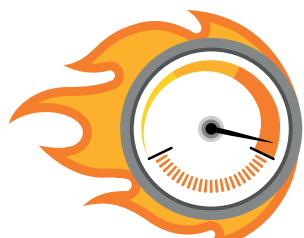
C safe-error attacks against protected implementations to attack ECDSA[1].

[1] Fouque et al., "Safe-Errors on SPA Protected Implementations with the Atomicity Technique"; Dubeuf, Hely, and Beroulle, "Enhanced Elliptic Curve Scalar Multiplication Secure Against Side Channel Attacks and Safe Errors."

In this presentation

We extend the previous results, and show that a C safe-error attack is also possible on these implementations:

- Assembly optimized implementation of P-256:
  - OpenSSL since version 1.0.2
  - BoringSSL
  - LibreSSL/OpenBSD
- P-224, P-384 and P-521 in BoringSSL



Plan

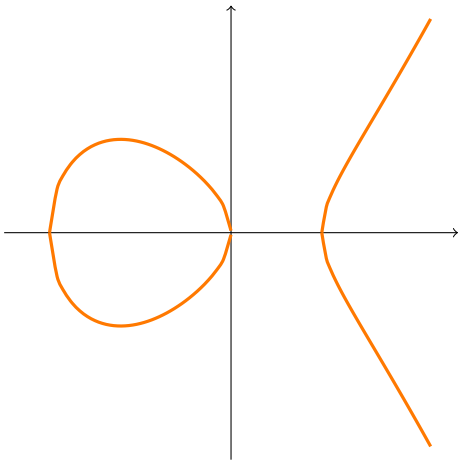
1 Why dummy codes in ECC?

2 Presentation of the attack

3 Why it works

4 Mitigations and conclusion

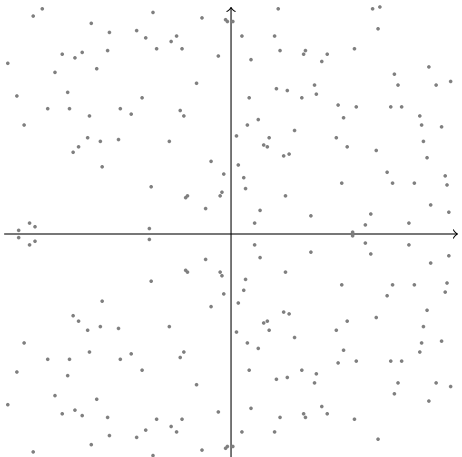
Operations on elliptic curves



Basic operations:

- Addition: $P + Q$
- Doubling: $P + P = 2P$
- $P + \mathcal{O} = \mathcal{O} + P = P$

Operations on elliptic curves

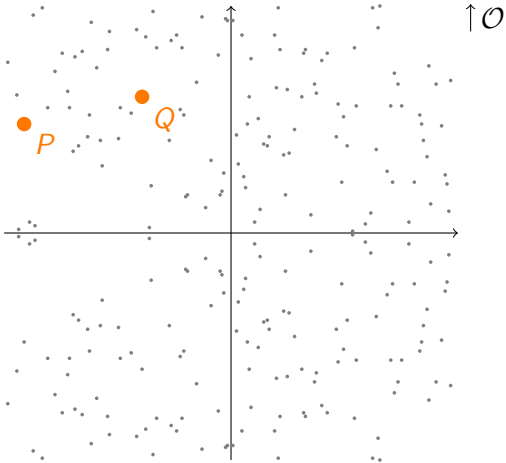


$E : y^2 = x^3 - 3x + 137 \mod 251$

Basic operations:

- Addition: $P + Q$
- Doubling: $P + P = 2P$
- $P + \mathcal{O} = \mathcal{O} + P = P$

Operations on elliptic curves

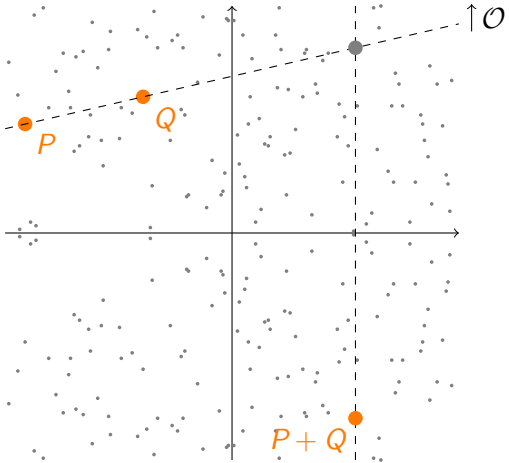


$E : y^2 = x^3 - 3x + 137 \mod 251$

Basic operations:

- Addition: $P + Q$
- Doubling: $P + P = 2P$
- $P + \mathcal{O} = \mathcal{O} + P = P$

Operations on elliptic curves



$E : y^2 = x^3 - 3x + 137 \mod 251$

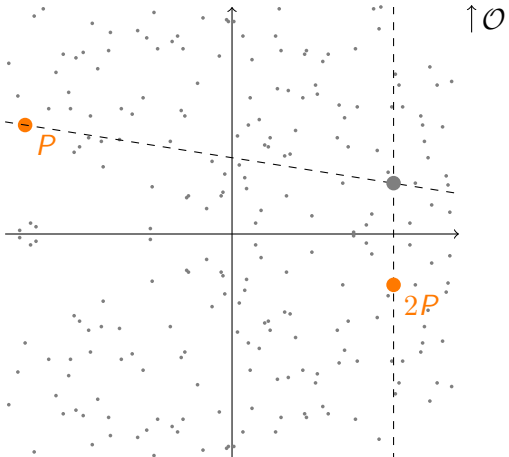
Basic operations:

- Addition: $P + Q$
- Doubling: $P + P = 2P$
- $P + \mathcal{O} = \mathcal{O} + P = P$

$$\lambda = \frac{y_P - y_Q}{x_P - x_Q}$$

$$\begin{cases} x_{P+Q} = \lambda^2 - x_P - x_Q \\ y_{P+Q} = \lambda(x_P - x_{P+Q}) - x_P \end{cases}$$

Operations on elliptic curves



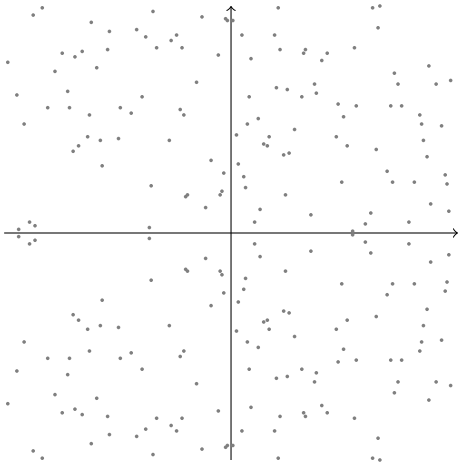
Basic operations:

- Addition: $P + Q$
- Doubling: $P + P = 2P$
- $P + \mathcal{O} = \mathcal{O} + P = P$

$$\lambda = \frac{3x_P + a}{2y_P}$$

$$\begin{cases} x_{2P} = \lambda^2 - 2x_P \\ y_{2P} = \lambda(x_P - x_{2P}) - x_P \end{cases}$$

Operations on elliptic curves



$E : y^2 = x^3 - 3x + 137 \mod 251$

Basic operations:

- Addition: $P + Q$
- Doubling: $P + P = 2P$
- $P + \mathcal{O} = \mathcal{O} + P = P$

Scalar multiplication:

$$kP = P + \cdots + P$$

- Discrete logarithm problem: hard to find k from P and kP
- k is often secret (private key or nonce)

Windowing scalar multiplication

- Available operations: point addition (A), point doubling (D)
- For efficiency: split k in groups of consecutive bits (windows)

Example: $k = 232 = (11\,101\,000)_2$

$$= \qquad \mathcal{O}$$

Windowing scalar multiplication

- Available operations: point addition (A), point doubling (D)
- For efficiency: split k in groups of consecutive bits (windows)

Example: $k = 232 = (11\,101\,000)_2$

11	$3P$	$=$	$3P$

Historic of operations:

Windowing scalar multiplication

- Available operations: point addition (A), point doubling (D)
- For efficiency: split k in groups of consecutive bits (windows)

Example: $k = 232 = (11\,\underline{101}\,000)_2$

11	$3P$	$=$	$3P$	
$11\,000$	$2^3 \cdot 3P$	$=$	$24P$	$D\,D\,D$

Historic of operations: D D D

Windowing scalar multiplication

- Available operations: point addition (A), point doubling (D)
- For efficiency: split k in groups of consecutive bits (windows)

Example: $k = 232 = (11\,101\,000)_2$

11	$3P$	$=$	$3P$	
$11\,000$	$2^3 \cdot 3P$	$=$	$24P$	$D\,D\,D$
$11\,101$	$24P + 5P$	$=$	$29P$	A

Historic of operations: D D D A

Windowing scalar multiplication

- Available operations: point addition (A), point doubling (D)
- For efficiency: split k in groups of consecutive bits (windows)

Example: $k = 232 = (11\,101\,\underline{000})_2$

11	$3P$	$=$	$3P$	
$11\,000$	$2^3 \cdot 3P$	$=$	$24P$	$D\,D\,D$
$11\,101$	$24P + 5P$	$=$	$29P$	A
$11\,101\,000$	$2^3 \cdot 29P$	$=$	$232P$	$D\,D\,D$

Historic of operations: D D D A D D D

Windowing scalar multiplication

- Available operations: point addition (A), point doubling (D)
- For efficiency: split k in groups of consecutive bits (windows)

Example: $k = 232 = (11\,101\,000)_2$

11	$3P$	$=$	$3P$	
$11\,000$	$2^3 \cdot 3P$	$=$	$24P$	$D\,D\,D$
$11\,101$	$24P + 5P$	$=$	$29P$	A
$11\,101\,000$	$2^3 \cdot 29P$	$=$	$232P$	$D\,D\,D$
$11\,101\,000$	$232P$	$=$	$232P$	

Historic of operations: D D D A D D D

Windowing scalar multiplication

- Available operations: point addition (A), point doubling (D)
- For efficiency: split k in groups of consecutive bits (windows)

Example: $k = 232 = (11\,101\,000)_2$

11	$3P$	$=$	$3P$	
$11\,000$	$2^3 \cdot 3P$	$=$	$24P$	$D\,D\,D$
$11\,101$	$24P + 5P$	$=$	$29P$	A
$11\,101\,000$	$2^3 \cdot 29P$	$=$	$232P$	$D\,D\,D$
$11\,101\,000$	$232P$	$=$	$232P$	

Historic of operations: D D D A D D D

- From power consumption, attacker remarks the missing addition and learns that:

$$k = ** \; *** 000$$

Windowing scalar multiplication

- Available operations: point addition (A), point doubling (D)
- For efficiency: split k in groups of consecutive bits (windows)

Example: $k = 232 = (11\,101\,000)_2$

11	$3P$	$=$	$3P$	
$11\,000$	$2^3 \cdot 3P$	$=$	$24P$	$D\,D\,D$
$11\,101$	$24P + 5P$	$=$	$29P$	A
$11\,101\,000$	$2^3 \cdot 29P$	$=$	$232P$	$D\,D\,D$
$11\,101\,000$	$232P$	$=$	$232P$	

Historic of operations: D D D A D D D

- From power consumption, attacker remarks the missing addition and learns that:

$$k = ** \; *** 000$$

- Solution: perform a dummy point addition:

D D D A D D D A

Windowing scalar multiplication

- Available operations: point addition (A), point doubling (D)
- For efficiency: split k in groups of consecutive bits (windows)

Example: $k = 232 = (11\,101\,000)_2$

11	$3P$	$=$	$3P$	
$11\,000$	$2^3 \cdot 3P$	$=$	$24P$	$D\,D\,D$
$11\,101$	$24P + 5P$	$=$	$29P$	A
$11\,101\,000$	$2^3 \cdot 29P$	$=$	$232P$	$D\,D\,D$
$11\,101\,000$	$232P$	$=$	$232P$	

Historic of operations: D D D A D D D

- From power consumption, attacker remarks the missing addition and learns that:

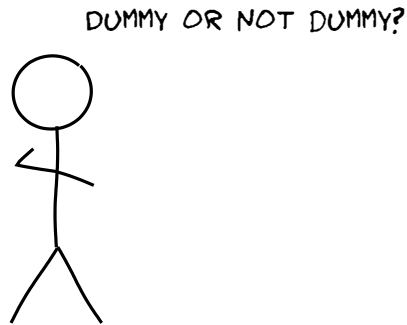
$$k = ** \; *** 000$$

- Solution: perform a dummy point addition:

D D D A D D D A

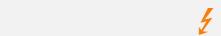
- Consequence: same sequence of operations for all possible secret k

Principle of C safe-error



Principle of C safe-error

Make a fault in the last point addition (exact details in the article):

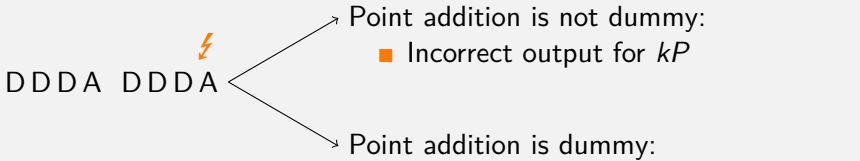


Principle of C safe-error

Make a fault in the last point addition (exact details in the article):

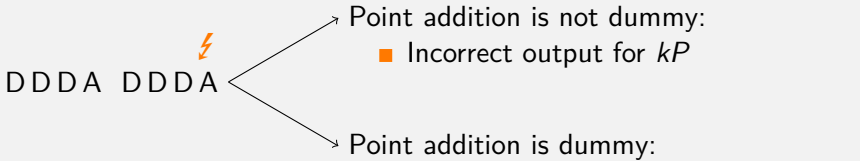
Principle of C safe-error

Make a fault in the last point addition (exact details in the article):



Principle of C safe-error

Make a fault in the last point addition (exact details in the article):



Number ℓ of bits of the last window in the targeted implementations:

- Assembly optimized implementation of P-256: 5 most significant bits
- BoringSSL (P-224, P-384, P-521): 5 least significant bits

Plan

1 Why dummy codes in ECC?

2 Presentation of the attack

3 Why it works

4 Mitigations and conclusion

Main steps of the attack

Objective: retrieve an ECDSA private key

Prerequisite:

- Physical access to the device
- Can inject a fault on potential dummy addition
- Acquire public data (public key, signatures, messages)

Steps:

[2] `https://github.com/orangecertcc/ecdummy` (MIT license)

Main steps of the attack

Objective: retrieve an ECDSA private key

Prerequisite:

- Physical access to the device
- Can inject a fault on potential dummy addition
- Acquire public data (public key, signatures, messages)

Steps:

1. Make a fault in last point addition of ECDSA signature calculation (random computational error is sufficient)

[2] `https://github.com/orangecertcc/ecdummy` (MIT license)

Main steps of the attack

Objective: retrieve an ECDSA private key

Prerequisite:

- Physical access to the device
- Can inject a fault on potential dummy addition
- Acquire public data (public key, signatures, messages)

Steps:

1. Make a fault in last point addition of ECDSA signature calculation (random computational error is sufficient)
2. Keep the signature only if valid
3. Repeat the above steps

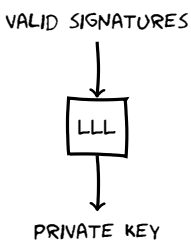
[2] `https://github.com/orangecertcc/ecdummy` (MIT license)

Main steps of the attack

Objective: retrieve an ECDSA private key

Prerequisite:

- Physical access to the device
- Can inject a fault on potential dummy addition
- Acquire public data (public key, signatures, messages)



Steps:

1. Make a fault in last point addition of ECDSA signature calculation (random computational error is sufficient)
2. Keep the signature only if valid
3. Repeat the above steps
4. Use our tool[2] to recover the private key from valid signatures.

[2] `https://github.com/orangecertcc/ecdummy` (MIT license)

Performance

Minimum number of valid signatures to recover the private key:

Number of bits ℓ of last window		4	5	6	7
Elliptic curve size	224-bit	56	45	37	31
	256-bit	65	52	43	36
	384-bit		91	65	56

Performance

Minimum number of valid signatures to recover the private key:

Number of bits ℓ of last window		4	5	6	7
Elliptic curve size	224-bit	56	45	37	31
	256-bit	65	52	43	36
	384-bit		91	65	56

Average of one valid signature out of 2^ℓ signatures attacked

Example for curve P-256 in OpenSSL ($\ell = 5$) out of 100 tests on average

- 54-55 valid signatures
- 1764 signatures attacked

Tools for the attack

Tools for the attack in script `ec.py`:

- `check_signature(curve, pubkey_point, signature)`
- `findkey(curve, pubkey_point, valid_signatures, msb, l)`
  - `l`: number of bits of last window
  - `msb`: last window corresponds to most or least significant bits

Demonstration of the attack

Target: assembly optimized implementation of P-256 in OpenSSL 1.1.1g

- Code modified to simulate the fault

```
for (i = 1; i < 37; i++) {
  //(...)
  if (i == 36) {
    ecp_nistz256_point_add_affine_faulty(&p.p, &p.p, &t.a);
  }
  else {
    ecp_nistz256_point_add_affine(&p.p, &p.p, &t.a);
  }
}
```

- Last window: 5 most significant bits
- The tool will be called as

  `findkey(secp256r1, pubkey_point, valid_signatures, True, 5)`

Demonstration of the attack

Target: assembly optimized implementation of P-256 in OpenSSL 1.1.1g

- Code modified to simulate the fault

```
for (i = 1; i < 37; i++) {
  //(...)
  if (i == 36) {
    ecp_nistz256_point_add_affine_faulty(&p.p, &p.p, &t.a);
  }
  else {
    ecp_nistz256_point_add_affine(&p.p, &p.p, &t.a);
  }
}
```

```
ecp_nistz256_sqr_mont(Z1sqr, in1_z);
Z1sqr[0] ^= 123456789; // "random" fault
```

- Last window: 5 most significant bits
- The tool will be called as

```
findkey(secp256r1, pubkey_point, valid_signatures, True, 5)
```

Plan

1 Why dummy codes in ECC?

2 Presentation of the attack

3 Why it works

4 Mitigations and conclusion

ECDSA

Given a private key d in $[1, q-1]$, the process of signing a file is:

- $m \leftarrow$ hash of the file
- $k \leftarrow$ random secret nonce in $[1, q-1]$

$$\text{signature: } \begin{cases} r & = \\ s & = \end{cases}$$

[3] Nguyen and Shparlinski, "The Insecurity of the Elliptic Curve Digital Signature Algorithm with Partially Known Nonces."

ECDSA

Given a private key d in $[1, q-1]$, the process of signing a file is:

- $m \leftarrow$ hash of the file
- $k \leftarrow$ random secret nonce in $[1, q-1]$

$$\text{signature: } \begin{cases} r &= x(kP) \\ s &= (dr+m)/k \end{cases}$$

[3] Nguyen and Shparlinski, "The Insecurity of the Elliptic Curve Digital Signature Algorithm with Partially Known Nonces."

ECDSA

Given a private key d in $[1, q-1]$, the process of signing a file is:

- $m \leftarrow$ hash of the file
- $k \leftarrow$ random secret nonce in $[1, q-1]$

$$\text{signature:} \begin{cases} r &= x(kP) \\ s &= (dr+m)/k \end{cases}$$

3-> d can be recovered from partial knowledge of k for several signatures[3]

[3] Nguyen and Shparlinski, "The Insecurity of the Elliptic Curve Digital Signature Algorithm with Partially Known Nonces."

Idea of the attack on ECDSA

We can rewrite the signature:

$$d \cdot r/s + m/s = \boxed{k}$$

Idea of the attack on ECDSA

We can rewrite the signature:

$$d \cdot r/s + m/s =$$

Idea of the attack on ECDSA

We can rewrite the signature:

$$d \cdot r/s + m/s = \underbrace{\qquad\qquad}_{\text{unknown (“small”)}}$$

Idea of the attack on ECDSA

We can rewrite the signature:

$$d \cdot r/s + m/s = \underbrace{\qquad\qquad}_{\text{unknown (“small”)}}$$

$$d \cdot u_1 \quad + \quad v_1 \quad = \quad \texttt{small}$$

Idea of the attack on ECDSA

We can rewrite the signature:

$$d \cdot r/s + m/s = \underbrace{\qquad\qquad}_{\text{unknown (“small”)}}$$

$$\begin{array}{lllll} d \cdot u_1 & + & v_1 & = & \texttt{small} \\ d \cdot u_2 & + & v_2 & = & \texttt{small} \end{array}$$

Idea of the attack on ECDSA

We can rewrite the signature:

$$d \cdot r/s + m/s = \underbrace{\qquad\qquad}_{\text{unknown (“small”)}}$$

$$\begin{array}{lcll} d \cdot u_1 & + & v_1 & = \texttt{small} \\ d \cdot u_2 & + & v_2 & = \texttt{small} \\ & \vdots & & \\ d \cdot u_n & + & v_n & = \texttt{small} \end{array}$$

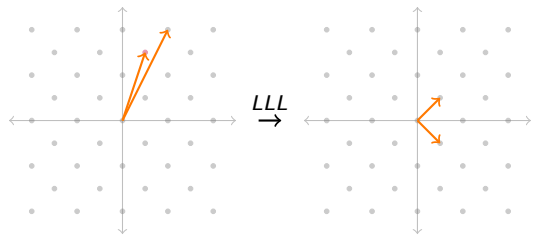
Idea of the attack on ECDSA

We can rewrite the signature:

$$d \cdot r/s + m/s = \boxed{\phantom{\text{unknown (“small”)}}}$$

unknown (“small”)

$$\begin{array}{lcll} d \cdot u_1 & + & v_1 & = \texttt{small} \\ d \cdot u_2 & + & v_2 & = \texttt{small} \\ & \vdots & & \\ d \cdot u_n & + & v_n & = \texttt{small} \end{array}$$



- LLL: find short vectors

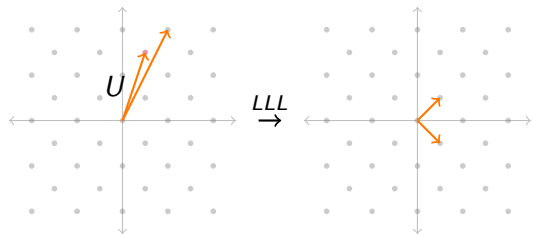
Idea of the attack on ECDSA

We can rewrite the signature:

$$d \cdot r/s + m/s = \boxed{\phantom{\text{unknown (“small”)}}}$$

unknown (“small”)

$$\begin{array}{llllll} d \cdot \boxed{u_1} & + & v_1 & = & \texttt{small} \\ d \cdot u_2 & + & v_2 & = & \texttt{small} \\ & \vdots & & & \\ d \cdot u_n & + & v_n & = & \texttt{small} \\ U & & & & \end{array}$$



- LLL: find short vectors

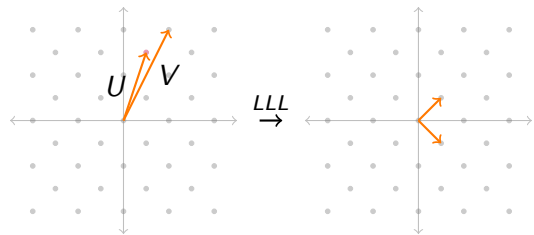
Idea of the attack on ECDSA

We can rewrite the signature:

$$d \cdot r/s + m/s = \boxed{\phantom{\text{unknown (“small”)}}}$$

unknown (“small”)

$$\begin{array}{ccccc} d \cdot u_1 & + & v_1 & = & \texttt{small} \\ d \cdot u_2 & + & v_2 & = & \texttt{small} \\ & & \vdots & & \\ d \cdot u_n & + & v_n & = & \texttt{small} \\ U & & V & & \end{array}$$



- LLL: find short vectors

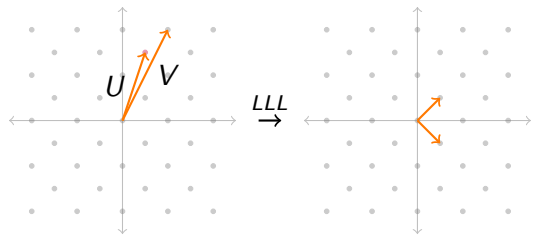
Idea of the attack on ECDSA

We can rewrite the signature:

$$d \cdot r/s + m/s = \boxed{\phantom{\text{unknown (“small”)}}}$$

unknown (“small”)

$$\begin{array}{ccccc}
d \cdot \boxed{u_1} & + & \boxed{v_1} & = & \texttt{small} \\
d \cdot \boxed{u_2} & + & \boxed{v_2} & = & \texttt{small} \\
 & & \vdots & & \\
d \cdot \boxed{u_n} & + & \boxed{v_n} & = & \texttt{small} \\
d \quad U & + & V & = &
\end{array}$$



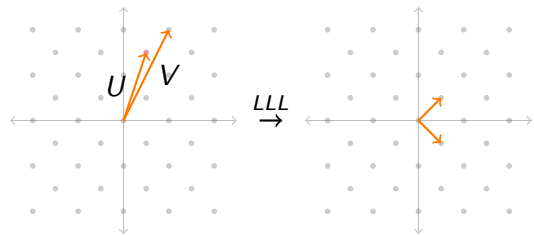
- LLL: find short vectors

Idea of the attack on ECDSA

We can rewrite the signature:

$$d \cdot r/s + m/s = \underbrace{\qquad\qquad}_{\text{unknown (“small”)}}$$

$$\begin{array}{ccccc}
d \cdot u_1 & + & v_1 & = & \texttt{small} \\
d \cdot u_2 & + & v_2 & = & \texttt{small} \\
 & \vdots & & & \\
d \cdot u_n & + & v_n & = & \texttt{small} \\
d\ U & + & V & = & \text{short vector}
\end{array}$$



- LLL: find short vectors
- If the short vector is found, we get d

Plan

1 Why dummy codes in ECC?

2 Presentation of the attack

3 Why it works

4 Mitigations and conclusion

Mitigations

Mitigations:

- Scalar encoding to avoid null windows
- Scalar blinding
- Avoid these cryptographic libraries for IoT devices

Conclusion

Wrap-up:

- Physical attack on ECDSA in OpenSSL and its forks
- Private key recovered from a few thousands signatures
- Proof of concept and tools for the attack available[4]
- Open questions: are there other libraries using dummy additions?

[4] `https://github.com/orangecertcc/ecdummy` (MIT license)