libecc: A Flexible Open-source ECC Library for Embedded Devices

Séminaire Sécurité des Systèmes Électroniques Embarqués

Ryad Benadjila, Arnaud Ebalard
ANSSI

<prenom.nom@ssi.gouv.fr>

14 February 2020
Presentation of a libecc

Context of development, process
Layout of the presentation

- Presentation of a libecc
  - Context of development, process

- Architecture of the library
  - Rationale and layers
  - Coding style and efforts
Layout of the presentation

- Presentation of a libecc
  - Context of development, process

- Architecture of the library
  - Rationale and layers
  - Coding style and efforts

- Discussion about security aspects
  - Software security
  - Side-Channel and fault attacks countermeasures
Layout of the presentation

- Presentation of a **libecc**
  - **Context** of development, **process**

- **Architecture** of the library
  - **Rationale** and **layers**
  - Coding style and **efforts**

- Discussion about **security** aspects
  - **Software security**
  - **Side-Channel** and **fault** attacks countermeasures

- **Usage** in projects
  - **Typical examples** and **advantages**
Context of libecc development
Why libecc?

Need for a **multi-platform** and **multi-arch** ECC library with:

- Code *simplicity* and *auditability* (static analysis and so on)
- Flexibility and portability
- Test vectors for various *signature algorithms* and *curves* 
  (e.g. for testing interoperability of IPsec stacks)
### Why libecc?

Why not mainstream libraries? (OpenSSL, GnuTLS, mbedTLS, ...)

- Generally only EC-DSA support
- Very constrained devices (Cortex-M0 and so on)
- State of the art? Still CVE prone:

![CVE Details](image)

- Fixed in OpenSSL 1.1.1d ([git commit](https://github.com/openssl/openssl/pull/7498)) (Affected 1.1.1-1.1.1c)
- Fixed in OpenSSL 1.1.1f ([git commit](https://github.com/openssl/openssl/pull/7502)) (Affected 1.1.0-1.1.0k)
- Fixed in OpenSSL 1.0.2t ([git commit](https://github.com/openssl/openssl/pull/7502)) (Affected 1.0.2-1.0.2s)
Why libecc?

Why not mainstream libraries? (OpenSSL, GnuTLS, mbedTLS, ...)

- Generally only EC-DSA support
- Very constrained devices (Cortex-M0 and so on)
- State of the art? Still CVE prone:
**What is libecc?**

- **Elliptic Curves** targeting **signatures** of ISO14888-3 and **versatile curves**:
  
  - EC-\{KC,G,R,S,FS\}DSA
  
  - SECP\{256,384,521\}r1, BrainpoolP\{224,256,384,512\}r1, ...

- Also exposes **APIs** to work with:
  
  - Natural numbers \(N\)
  
  - Finite fields over primes \(\mathbb{F}_p\)
  
  - Elliptic curves group arithmetic: ADD, DBL, ...

- Written in pure **C99** with **simplicity** and **portability** in mind

- Available on **ANSSI github**: [https://github.com/ANSSI-FR/libecc](https://github.com/ANSSI-FR/libecc) (dual BSD / GPLv2+ license)
Some limitations

- Only curves over $\mathbb{F}_p$ in the Weierstraß form: $y^2 = x^3 + a \times x + b$

- Means:
  - No support of curves over binary fields $\mathbb{F}_{2^m}$
  - No Edwards curves (i.e. no Ed25519 and so on)

- Simple arithmetic: no specific pseudo-Mersenne optimizations (NIST curves)

- Pure C $\Rightarrow$ constant time can be hard to achieve
  - More on this later

- No X.509 and ASN.1 handling (for the better!)
  - Library inputs and outputs use very simple custom formats
  - Hence some interoperability limitations
Some limitations

- Only curves over $\mathbb{F}_p$ in the Weierstraß form: $y^2 = x^3 + ax + b$

- Means:
  - No support of curves over binary fields $\mathbb{F}_{2^m}$
  - No Edwards curves (i.e. no Ed25519 and so on)

- Simple arithmetic: no specific pseudo-Mersenne optimizations (NIST curves)
  - Pure C $\Rightarrow$ constant time can be hard to achieve
  - More on this later

- No X.509 and ASN.1 handling (for the better!)
  - Library inputs and outputs use very simple custom formats
  - Hence some interoperability limitations

Frama-C validated X.509 parser, free of RunTime Errors:
https://github.com/ANSSI-FR/x509-parser
Library internals: \textit{architecture} and \textit{code}
LIBECC ARCHITECTURE

Machine related

words size
(16, 32, 64 bits)
C optimizations
**LIBECC ARCHITECTURE**

- **Natural Numbers \( \mathbb{N} \) arithmetic**
  - (modular) add, sub
  - Montgomery mul
  - Euclidean div, mod inv

- **Machine related**
  - words size
    - (16, 32, 64 bits)
  - C optimizations
LIBECC ARCHITECTURE

**Finite Fields** $\mathbb{F}_p$ arithmetic:
- add, sub, mul, inv
- Power

**Natural Numbers** $\mathbb{N}$ arithmetic:
- (modular) add, sub
- Montgomery mul
- Euclidean div, mod inv

**Machine related**:
- words size (16, 32, 64 bits)
- C optimizations
LIBECC ARCHITECTURE

Curves parameters
- SECP, Brainpool, ...
- Parameters ($\mathbb{F}_p$, order)

Elliptic Curves arithmetic
- ADD, DBL
- Scalar mul
- affine/projective

Finite Fields $\mathbb{F}_p$ arithmetic
- add, sub, mul, inv
- Power

Natural Numbers $\mathbb{N}$ arithmetic
- (modular) add, sub
- Montgomery mul
- Euclidean div, mod inv

Machine related
- words size (16, 32, 64 bits)
- C optimizations

Libecc architecture
- Words size
- C optimizations
- Machine related
- Natural Numbers $\mathbb{N}$ arithmetic
- Finite Fields $\mathbb{F}_p$ arithmetic
- Elliptic Curves arithmetic
- Curves parameters

Libecc conf files
- Tests (vectors, perf, …)
- External deps
- Utils (buffers copy, compare, …)
- Scripts

Libecc documentation
- Auxiliary modules
- Hash algorithms (SHA-2, SHA-3)
- Signatures (EC{*}DSA (ISO14888-3))
- Curves parameters
- Parameters ($\mathbb{F}_p$, order)

Libecc source code
- C: 141 .c/.h files, 18295 SLOC (89.22%)
- Python: 3 files, 1913 SLOC (9.33%)
- Bash: 4 files, 297 SLOC (1.45%)
LIBECC ARCHITECTURE

- **Signatures**
  - EC{*}DSA (ISO14888-3)
    - sign, verify

- **Hash functions**
  - Hash algorithms (SHA-2, SHA-3)

- **Elliptic Curves arithmetic**
  - ADD, DBL
  - Scalar mul
  - affine/projective

- **Finite Fields \( \mathbb{F}_p \) arithmetic**
  - add, sub, mul, inv
  - Power

- **Natural Numbers \( \mathbb{N} \) arithmetic**
  - (modular) add, sub
  - Montgomery mul
  - Euclidean div, mod inv

- **Curves parameters**
  - SECP, Brainpool, ...
  - Parameters (\( \mathbb{F}_p \), order)

- **Machine related**
  - words size (16, 32, 64 bits)
  - C optimizations

- **Library internals**

- **Security**

- **Projects**

- **Conclusion**

---

libecc - SemSecuElec february 2020 - Rennes
LIBECC ARCHITECTURE

- **Elliptic Curves arithmetic**
  - ADD, DBL
  - Scalar mul
  - affine/projective

- **Finite Fields \( \mathbb{F}_p \) arithmetic**
  - add, sub, mul, inv
  - Power

- **Natural Numbers \( \mathbb{N} \) arithmetic**
  - (modular) add, sub
  - Montgomery mul
  - Euclidean div, mod inv

- **Curves parameters**
  - SECP, Brainpool, ...
  - Parameters \((\mathbb{F}_p, \text{order})\)

- **Signatures**
  - EC\(*)\)-DSA (ISO14888-3)
  - sign, verify

- **Hash functions**
  - Hash algorithms (SHA-2, SHA-3)

- **Auxiliary modules**
  - libecc conf files
  - Tests (vectors, perf, ...)
  - External deps
  - Utils (buffers copy, compare, ...)
  - Scripts

- **Machine related**
  - words size
    - (16, 32, 64 bits)
  - C optimizations
C: 141 .c/.h files, 18295 SLOC (89.22%)

Python: 3 files, 1913 SLOC (9.33%)

Bash: 4 files, 297 SLOC (1.45%)
libecc - SemSecuElec february 2020 - Rennes

LIBECC CONFIGURATION FILE

$ cat lib_ecc_config.h

#ifndef __LIB_ECC_CONFIG_H__
#define __LIB_ECC_CONFIG_H__

/* This configuration file provides
various knobs to configure what
will be built in the library
(supported curves, hash algorithms
and signature/verification schemes).
*/

/* Supported curves */
#define WITH_CURVE_FRP256V1
#define WITH_CURVE_SECP192R1
#define WITH_CURVE_SECP224R1
#define WITH_CURVE_SECP256R1
#define WITH_CURVE_SECP384R1
#define WITH_CURVE_SECP521R1
#define WITH_CURVE_BRAINPOOLP224R1
#define WITH_CURVE_BRAINPOOLP256R1
#define WITH_CURVE_BRAINPOOLP384R1
#define WITH_CURVE_BRAINPOOLP512R1

/* Supported hash algorithms */
#define WITH_HASH_SHA224
#define WITH_HASH_SHA256
#define WITH_HASH_SHA384
#define WITH_HASH_SHA512
#define WITH_HASH_SHA3_224
#define WITH_HASH_SHA3_256
#define WITH_HASH_SHA3_384
#define WITH_HASH_SHA3_512

/* Supported sig/verifschemes */
#define WITH_SIG_ECDSA
#define WITH_SIG_ECKCDSA
#define WITH_SIG_ECSDSA
#define WITH_SIG_ECGSDSA
#define WITH_SIG_ECGDSA
#define WITH_SIG_ECRDSA

#define WITH_CURVE_GOST256
#define WITH_CURVE_GOST512

/* Supported hash algorithms */
#define WITH_HASH_SHA224
#define WITH_HASH_SHA256
#define WITH_HASH_SHA384
#define WITH_HASH_SHA512
#define WITH_HASH_SHA3_224
#define WITH_HASH_SHA3_256
#define WITH_HASH_SHA3_384
#define WITH_HASH_SHA3_512

#define WITH_CURVE_GOST256
#define WITH_CURVE_GOST512

/* Supported sig/verifschemes */
#define WITH_SIG_ECDSA
#define WITH_SIG_ECKCDSA
#define WITH_SIG_ECSDSA
#define WITH_SIG_ECGSDSA
#define WITH_SIG_ECGDSA
#define WITH_SIG_ECRDSA

#endif /* __LIB_ECC_CONFIG_H__ */
Development leit motiv: code **simplicity**, **readability** and **auditability**

- Pure **C99**, as **strong typing** as possible
- **Security** before **performance**
- **Pre** and **post-conditions** in functions
- **Magic values** for structures **initialization**
- **Configurable** machine words **size**
- No **dynamic allocation**: **static buffers** on the stack
- **Constant time** as much as possible (in C, more on this later)
Portability: compilation flexibility

- Only two C99 critical necessary features:
  - `long long int` types support
  - Designated initializers for structures
  - Compatibility with non GNU toolchains (Keil, ...)

⇒ Portability on many platforms: x86, arm, MIPS, PowerPC, AVR, ...
Portability: compilation flexibility

- Only two C99 critical necessary features:
  - long long int types support
  - Designated initializers for structures
  - Compatibility with non GNU toolchains (Keil, ...)

  ⇒ Portability on many platforms: x86, arm, MIPS, PowerPC, AVR, ...

- Makefile allows to override many elements:
  - Compiler, {C,LD}FLAGS
  - Standard library usage, WORDSIZE word size

$ CC=sdcc AR=sdar RANLIB=sdranlib CFLAGS="--std-sdcc99 -DWORDSIZE=16" LDFLAGS=" " make

Zilog Z80 variant (8-bit CPU)
Tests

- Non-regression tests
  - Python generated tests for low level $\mathbb{N}$ and $\mathbb{F}_p$ arithmetic
# Tests

## Non-regression tests

- **Python** generated tests for low level $\mathbb{N}$ and $\mathbb{F}_p$ arithmetic
- **Standard test vectors** for EC{*}DSA
- **Randomized test vectors**
  (random keys and check that signature and verification match)

```
$ ./build/ec_self_tests

======= Known test vectors test =========
[+] ECDSA-SHA224/secp224r1 selftests: known test vectors sig/verif ok
[+] ECDSA-SHA256/secp256r1 selftests: known test vectors sig/verif ok
...
[+] ECRDSA-SHA256/GOST-256-curve selftests: known test vectors sig/verif ok
[+] ECRDSA-SHA512/GOST-512-curve selftests: known test vectors sig/verif ok

======= Random sig/verif test ==========
[+] ECDSA-SHA224/FRP256V1 randtests: random import/export with sig/verif ok
[+] ECDSA-SHA224/SECP192R1 randtests: random import/export with sig/verif ok
...
[+] ECRDSA-SHA3_512/GOST256 randtests: random import/export with sig/verif ok
[+] ECRDSA-SHA3_512/GOST512 randtests: random import/export with sig/verif ok

======= Performance test ==========
[+] ECDSA-SHA224/FRP256V1 perf: 350 sign/s and 180 verif/s
[+] ECDSA-SHA224/SECP192R1 perf: 500 sign/s and 250 verif/s
```
Non-regression tests

- Python generated tests for low level $\mathbb{N}$ and $\mathbb{F}_p$ arithmetic
- Standard test vectors for EC{*}DSA
- Randomized test vectors (random keys and check that signature and verification match)
- Cross-build (docker) and cross-run (qemu-static) scripts

⇒ Extensive ≈ 200 travis jobs matrix for non-regression
Big numbers representation

- Big numbers:
  - Primitive types (u8, u16, u32, u64) automatically handled or forced
Big numbers representation

- Big numbers:
  - **Primitive types** (u8, u16, u32, u64) automatically handled or forced

- **Input/output representation** is classical big endian

```
0x0102030405060708090a0b0c0d0e0f00 ... 0102030405060708090a0b0c0d0e0f00
```

```
01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f 00
```

```
buffer[0] 
```
Big numbers representation

Big numbers:

- **Internal representation** is:
  - Little endian array of words
  - Machine endianness inside words

```
0x0102030405060708090a0b0c0d0e0f00 ... 0102030405060708090a0b0c0d0e0f00
```

```
word[0]  word[1]  ...
```

```
08 07 06 05 04 03 02 01 00 0f 0e 0d 0c 0b 0a 09 ...
```

Native 64-bit unsigned long long
(x86_64, arm64, ...)

```
u64 word[0]  u64 word[1]
hword 0  hword 0
u32 low  u32 high
```
Big numbers representation

- **Big numbers:**
  - **Internal representation** is:
    - Little endian array of words
    - Machine endianness inside words
  - **Default big numbers maximum size** is on par with maximum curve size
  - **Big numbers size** is configurable to a certain extent (5300 bits for 64 bits words)
  - To avoid uncontrolled size growth, each big number has a dynamic size to trim
The core library has no dependency, not even libc

External dependencies:

- Critical:
  - Random source

- Critical for EC{*}DSA signature
  - nonce generation
The core library has no dependency, not even libc

External dependencies:

Critical:
  - Random source

Non-critical:
  - Printing function (for debug)
  - Timing primitive (for performance measurements)

When builds on known OSes (*NIX, Windows, MacOS) fallback to known sources (/dev/random, printf, ...)

External dependencies:

Critical for EC{*}DSA signature
nonce generation
LIBECC EXPANDABILITY

- Design choices (for security):
  - Supported Elliptic Curves are built at compilation time
    ⇒ No dynamic anonymous curves injection
  - APIs and structures made easy to add new signatures, hash functions, ...
LIBECC EXPANDABILITY

■ Design choices (for security):
  ➤ Supported Elliptic Curves are built at compilation time
    ⇒ No dynamic anonymous curves injection
  ➤ APIs and structures made easy to add new signatures, hash functions, ...

■ Python script [expand_libecc.py] to add new curves:
  ➤ A parallel pure Python ECC implementation
  ➤ Generates parameters, test vectors, ...

```bash
scripts/expand_libecc.py --name="YOURCURVENAME" --prime=... --order=... --a=... \ 
--b=... --gx=... --gy=... --cofactor=... --oid=THEOID --add-test-vectors=X
```

> name: name of the curve in the form of a string
> prime: prime number representing the curve prime field
> order: prime number representing the generator order
> cofactor: cofactor of the curve
> a: 'a' coefficient of the short Weierstrass equation of the curve
> b: 'b' coefficient of the short Weierstrass equation of the curve
> gx: x coordinate of the generator G
> gy: y coordinate of the generator G
> oid: optional OID of the curve
## LIBECC PERFORMANCE

- **Security and simplicity over performance:**
  - Decent performance
  - Decent memory footprint

<table>
<thead>
<tr>
<th>COMPARISON FOR BRAINPOOLP256R1</th>
<th>library</th>
<th>sign</th>
<th>verify</th>
<th>RAM/ROM</th>
</tr>
</thead>
<tbody>
<tr>
<td>CORE i7-5500U @ 2.40GHz</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>libecc</td>
<td>500/s</td>
<td>250/s</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>mbedTLS</td>
<td>426/s</td>
<td>106/s</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>OpenSSL</td>
<td>2463/s</td>
<td>1767/s</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>MARVELL ARMADA A388 @ 1.60GHz</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>libecc</td>
<td>55/s</td>
<td>28/s</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>mbedTLS</td>
<td>33/s</td>
<td>8/s</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>OpenSSL</td>
<td>369/s</td>
<td>332/s</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>CORTEX-M3 MCU @ 72MHz</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>libecc</td>
<td>0.91/s</td>
<td>0.45/s</td>
<td>8KB/32KB</td>
<td></td>
</tr>
<tr>
<td>mbedTLS</td>
<td>0.42/s</td>
<td>0.21/s</td>
<td>3KB/32KB</td>
<td></td>
</tr>
<tr>
<td>OpenSSL</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td></td>
</tr>
</tbody>
</table>

“Fair” comparison (no NIST optimizations)
**LIBECC PERFORMANCE**

- **But:**
  - Your platform may come with a **modular arithmetic accelerator** (with a Montgomery multiplier)
  - **libecc modularity** ⇒ switching from software implementation is **straightforward**
  - Provides a great **performance boost**

- Previously in **Eurisko project**, **ST33G1M2** running @25MHz:
  - **1500 ms** for software **ECDSA** speed (SHA256/FRP256)
  - **250 ms** when switching to **Nescrypt** (arithmetic accelerator)
  - A 700 lines patch
Security aspects: software, SCA, faults, ...
Software security

Code pattern for initialization and checks

```c
/* A structure representing an object */
typedef struct {
  ... /* Useful data */
  encapsulated_object obj; /* Encapsulated objects */
  word_t magic;
} math_object;

typedef math_object *math_object_t;
typedef const math_object *math_object_src_t;

void math_object_init(math_object_t A, ...) {
  MUST_HAVE(A != NULL);
  A->magic = MAGIC;
  encapsulated_object_init(&((A->obj));
  ...
}

void math_object_check_initialized(math_object_src_t A) {
  MUST_HAVE ((A != NULL) && (A->magic == MAGIC) && ...);
}

void math_object_uninit(math_object_t A) {
  math_object_check_initialized(A);
  /* Uninitialize encapsulated objects */
  encapsulated_object_uninit(&((A->obj));
  /* Uninitialize object */
  A->magic = WORD(0);
  /* Zeroize useful data */
  ...
}

/* Function operating on math object */
void function(math_object_src_t in1, math_object_src_t in2, math_object_t out) {
  /* Declare local variables */
  math_object tmp;
  ...
  /* Check input initialization */
  math_object_check_initialized(in1);
  math_object_check_initialized(in2);
  ...
  /* Handle aliasing, initialize output */
  if ((out != in1) && (out != in2)) {
    math_object_init(out, ...);
    ...
  }
  /* Algorithm core */
  ...
  /* Uninitialize local variables */
  math_object_uninit(&tmp);
  return;
}
```

Code pattern for initialization and checks

/* Function operating on math object */
void function(math_object_src_t in1, math_object_src_t in2, math_object_t out) {
  /* Declare local variables */
  math_object tmp;
  ...
  /* Check input initialization */
  math_object_check_initialized(in1);
  math_object_check_initialized(in2);
  ...
  /* Handle aliasing, initialize output */
  if ((out != in1) && (out != in2)) {
    math_object_init(out, ...);
    ...
  }
  /* Algorithm core */
  ...
  /* Uninitialize local variables */
  math_object_uninit(&tmp);
  return;
```
Software security

**Code pattern for initialization and checks**

```c
/* A structure representing an object */
typedef struct {
    ...
    /* Useful data */
    encapsulated_object obj; /* Encapsulated objects */
    word_t magic;
} math_object;

typedef math_object *math_object_t;
typedef const math_object *math_object_src_t;

void math_object_init(math_object_t A, ...)
{
    MUST_HAVE(A != NULL);
    A->magic = MAGIC;
    encapsulated_object_init(&(A->obj));
    ...
}

void math_object_check_initialized(math_object_src_t A)
{
    MUST_HAVE((A != NULL) && (A->magic == MAGIC) && ...);
}

void math_object_uninit(math_object_t A)
{
    math_object_check_initialized(A);
    /* Uninitialize encapsulated objects */
    encapsulated_object_uninit(&(A->obj));
    /* Uninitialize object */
    A->magic = WORD(0);
    /* Zeroize useful data */
    ...
}

/* Function operating on math object */
void function(math_object_src_t in1, math_object_src_t in2, math_object_t out)
{
    /* Declare local variables */
    math_object tmp;
    ...
    /* Check input initialization */
    math_object_check_initialized(in1);
    math_object_check_initialized(in2);
    ...
    /* Handle aliasing, initialize output */
    if ((out != in1) && (out != in2)) {
        math_object_init(out, ...);
    }
    /* Algorithm core */
    ...
    /* Uninitialize local variables */
    math_object_uninit(&tmp);
    return;
}
```

Code pattern for initialization and checks

/* Function operating on math object */
void function(math_object_src_t in1, math_object_src_t in2, math_object_t out)
{
    /* Declare local variables */
    math_object tmp;
    ...
    /* Check input initialization */
    math_object_check_initialized(in1);
    math_object_check_initialized(in2);
    ...
    /* Handle aliasing, initialize output */
    if ((out != in1) && (out != in2)) {
        math_object_init(out, ...);
    }
    /* Algorithm core */
    ...
    /* Uninitialize local variables */
    math_object_uninit(&tmp);
    return;
}

/* Function operating on math object */
void function(math_object_src_t in1, math_object_src_t in2, math_object_t out)
{
    /* Declare local variables */
    math_object tmp;
    ...
    /* Check input initialization */
    math_object_check_initialized(in1);
    math_object_check_initialized(in2);
    ...
    /* Handle aliasing, initialize output */
    if ((out != in1) && (out != in2)) {
        math_object_init(out, ...);
    }
    /* Algorithm core */
    ...
    /* Uninitialize local variables */
    math_object_uninit(&tmp);
    return;
}
Software security

Code pattern for initialization and checks

```c
/* A structure representing an object */
typedef struct {
    ... /* Useful data */
    encapsulated_object obj; /* Encapsulated objects */
    word_t magic;
} math_object;

typedef struct math_object_t;
typedef const math_object_t math_object_src_t;

void math_object_init(math_object_t A, ...)
{
    MUST_HAVE(A != NULL);
    A->magic = MAGIC;
    encapsulated_object_init(&A->obj);
    ...
}

void math_object_check_initialized(math_object_src_t A)
{
    MUST_HAVE ((A != NULL) && (A->magic == MAGIC) && ...);
}

void math_object_uninit(math_object_t A)
{
    math_object_check_initialized(A);
    /* Uninitialize encapsulated objects */
    encapsulated_object_uninit(&A->obj);
    /* Uninitialize object */
    A->magic = WORD(0);
    /* Zeroize useful data */
    ...
}

/* Function operating on math object */
void function(math_object_src_t in1, math_object_src_t in2,
              math_object_t out)
{
    /* Declare local variables */
    math_object tmp;
    ...
    /* Check input initialization */
    math_object_check_initialized(in1);
    math_object_check_initialized(in2);
    ...
    /* Handle aliasing, initialize output */
    if ((out != in1) && (out != in2)) {
        math_object_init(out, ...);
        ...
    }
    /* Algorithm core */
    ...
    /* Uninitialize local variables */
    math_object_uninit(&tmp);
    return;
}
```

Code pattern for initialization and checks

/* Function operating on math object */
void function(math_object_src_t in1, math_object_src_t in2,
              math_object_t out)
{
    /* Declare local variables */
    math_object tmp;
    ...
    /* Check input initialization */
    math_object_check_initialized(in1);
    math_object_check_initialized(in2);
    ...
    /* Handle aliasing, initialize output */
    if ((out != in1) && (out != in2)) {
        math_object_init(out, ...);
        ...
    }
    /* Algorithm core */
    ...
    /* Uninitialize local variables */
    math_object_uninit(&tmp);
    return;
```
Software security

Code pattern for initialization and checks

```c
/* A structure representing an object */
typedef struct {
    ...
    /* Useful data */
    encapsulated_object obj; /* Encapsulated objects */
    word_t magic;
} math_object;

typedef math_object *math_object_t;
typedef const math_object *math_object_src_t;

void math_object_init(math_object_t A, ...)
{
    MUST_HAVE(A != NULL);
    A->magic = MAGIC;
    encapsulated_object_init(&(A->obj));
    ...
}

void math_object_check_initialized(math_object_src_t A)
{
    MUST_HAVE((A != NULL) && (A->magic == MAGIC) && ...);
}

void math_object_uninit(math_object_t A)
{
    math_object_check_initialized(A);
    /* Uninitialize encapsulated objects */
    encapsulated_object_uninit(&(A->obj));
    /* Uninitialize object */
    A->magic = WORD(0);
    /* Zeroize useful data */
    ...
}

/* Function operating on math object */
void function(math_object_src_t in1, math_object_src_t in2, math_object_t out)
{
    /* Declare local variables */
    math_object tmp;
    ...
    /* Check input initialization */
    math_object_check_initialized(in1);
    math_object_check_initialized(in2);
    ...
    /* Handle aliasing, initialize output */
    if ((out != in1) && (out != in2)) {
        math_object_init(out, ...);
        ...
    }
    /* Algorithm core */
    ...
    /* Uninitialize local variables */
    math_object_uninit(&tmp);
    return;
}
```

Code pattern for initialization and checks

Software security
Software security

Code pattern for initialization and checks

/* A structure representing an object */
typedef struct {
    ... /* Useful data */
    encapsulated_object obj; /* Encapsulated objects */
    word_t magic;
} math_object;

typedef math_object *math_object_t;
typedef const math_object *math_object_src_t;

void math_object_init(math_object_t A, ...) {
    MUST_HAVE(A != NULL);
    A->magic = MAGIC;
    encapsulated_object_init(&A->obj);
    ...
}

void math_object_check_initialized(math_object_src_t A) {
    MUST_HAVE ((A != NULL) && (A->magic == MAGIC) && ...);
}

void math_object_uninit(math_object_t A) {
    math_object_check_initialized(A);
    /* Uninitialize encapsulated objects */
    encapsulated_object_uninit(&A->obj);
    /* Uninitialize object */
    A->magic = WORD(0);
    /* Zeroize useful data */
    ...
}

/* Function operating on math object */
void function(math_object_src_t in1, math_object_src_t in2, math_object_t out) {
    /* Declare local variables */
    math_object tmp;
    ...
    /* Check input initialization */
    math_object_check_initialized(in1);
    math_object_check_initialized(in2);
    ...
    /* Handle aliasing, initialize output */
    if ((out != in1) && (out != in2)) {
        math_object_init(out, ...);
        ...
    }
    /* Algorithm core */
    ...
    /* Uninitialize local variables */
    math_object_uninit(&tmp);
    return;
}
Software security

/* A structure representing an object */
typedef struct {
... /* Useful data */
    encapsulated_object obj; /* Encapsulated objects */
    word_t magic;
} math_object;

typedef math_object *math_object_t;
typedef const math_object *math_object_src_t;

void math_object_init(math_object_t A, ...)
{
    MUST_HAVE(A != NULL);
    A->magic = MAGIC;
    encapsulated_object_init(&(A->obj));
    ...
}

void math_object_check_initialized(math_object_src_t A)
{
    MUST_HAVE((A != NULL) && (A->magic == MAGIC));
    ...
}

void math_object_uninit(math_object_t A)
{
    math_object_check_initialized(A);
    /* Uninitialize encapsulated objects */
    encapsulated_object_uninit(&(A->obj));
    /* Uninitialize object */
    A->magic = WORD(0);
    /* Zeroize useful data */
    ...
}

/* Function operating on math object */
void function(math_object_src_t in1, math_object_src_t in2,
              math_object_t out)
{
    /* Declare local variables */
    ...  
    /* Check input initialization */
    math_object_check_initialized(in1);
    math_object_check_initialized(in2);
    ...
    /* Handle aliasing, initialize output */
    if ((out != in1) && (out != in2)) {
        math_object_init(out, ...);
        ...
    }
    /* Algorithm core */
    ...
    /* Uninitialize local variables */
    math_object_uninit(&tmp);
    return;

Static analysis whenever/wherever possible!
Passive attacks: Side-Channel based attacks

How:

⇒ Power, EM, Timing traces
ECC passive physical attacks

- **Passive attacks**: **Side-Channel** based attacks

  - **How**:
    - ⇒ **Power**, **EM**, **Timing** traces
    - ⇒ **Fine-grained local** (microarchitectural) attacks: **cache**, **branch prediction**, ...

![Image of a heatmap or tensor visualization with hexadecimal addresses from 0x7c680 to 0x7cc80.](image-url)
### ECC Passive Physical Attacks

- **Passive attacks**: Side-Channel based attacks
  - **How**:
    - ⇒ Power, EM, Timing traces
    - ⇒ Fine-grained local (microarchitectural) attacks: cache, branch prediction, ...
  - **Simple {Power, EM, …} Analysis**
    - ⇒ A single trace, recognize secret dependent patterns
    - ⇒ A single trace, extract secret MSB (or other bits)
Passive attacks: Side-Channel based attacks

How:
- Power, EM, Timing traces
- Fine-grained local (microarchitectural) attacks: cache, branch prediction, ...

Simple {Power, EM, ...} Analysis
- A single trace, recognize secret dependent patterns
- A single trace, extract secret MSB (or other bits)

Differential {Power, EM, ...} Analysis
- Multiple traces, vertical traces statistics
ECC passive physical attacks

- **Passive attacks**: Side-Channel based attacks
  - **How**:
    - Power, EM, Timing traces
    - Fine-grained local (microarchitectural) attacks: cache, branch prediction, ...
  - **Simple {Power, EM, ...} Analysis**
    - A single trace, recognize secret dependent patterns
    - A single trace, extract secret MSB (or other bits)
  - **Differential {Power, EM, ...} Analysis**
    - Multiple traces, vertical traces statistics
  - **Horizontal Collision Attacks**
    - A single trace, horizontal statistics on operands
**ECC SCA THREATS BIG PICTURE**

- EC{*}DSA signatures
  - Message $m$, Nonce $k$, priv key $x$
  - $W = k \times G = (W_x, W_y)$
  - $r = W_x||s = k^{-1}(x \times r + H(m))$
  - Use scalar mult

- EC scalar multiplication
  - $kP = k \times P$
  - $k = \{k_0, k_1, \ldots, k_n\} \Rightarrow \text{DBL/ADD}$

- EC Double and Add
  - $\text{ADD}(P, Q) / \text{DBL}(P)$
  - Point at infinity $O$ handling
  - Affine/Projective coordinates
  - Double/Add formulas

- $\mathbb{N}$ and $\mathbb{F}_p$
  - Big number operations
    - Size trimming
    - Division, inversion

---

**Message $m$, Nonce $k$, priv key $x$**

- $W = k \times G = (W_x, W_y)$
- $r = W_x||s = k^{-1}(x \times r + H(m))$
- Use scalar mult

- $kP = k \times P$
- $k = \{k_0, k_1, \ldots, k_n\} \Rightarrow \text{DBL/ADD}$

- $\text{ADD}(P, Q) / \text{DBL}(P)$
- Point at infinity $O$ handling
- Affine/Projective coordinates
- Double/Add formulas

- $\mathbb{N}$ and $\mathbb{F}_p$
- Big number operations
  - Size trimming
  - Division, inversion
**ECC SCA Threats Big Picture**

**EC{*}DSA signatures**
- Message $m$, Nonce $k$, priv key $x$
- $W = k \times G = (W_x, W_y)$
- $r = W_x \parallel s = k^{-1}(x \times r + H(m))$
- Find one $k$ with SPA $\Rightarrow$ recover $x$
- Collect partial information on multiple $k_i$ (MSB timing, SPA, ...)
- Use Hidden Number Problem $\Rightarrow$ recover $x$

**EC scalar multiplication**
- $kP = k \times P$
- $k = \{k_0, k_1, ..., k_n\} \Rightarrow$ DBL/ADD

**EC Double and Add**
- ADD$(P, Q) /$ DBL$(P)$
- Point at infinity $O$ handling
- Affine/Projective coordinates

**N and $\mathbb{F}_p$**
- Big number operations
- Size trimming
- Division, inversion

**Use scalar mult**
- Double and Add variants

**Single trace**
- Multiple traces

**Attack double and add variants with SPA, DPA, horizontal attacks $\Rightarrow$ recover $k$
- Single or multiple traces

**Distinguish ADD and DBL (non-unified formulas)**
- Distinguish point at infinity $O$ (incomplete formulas)
- Non-constant time operations (extra reductions, ...)
- Operands address/value usage collisions (horizontal attacks)

**Arithmetic blinding on $k$ and $x$:**
- $(b \times k) - 1 (x \times b \times r + b \times H(m))$

**Scalar multiplication blinding on $k$:**
- $k \times G = (k + b' \times q) \times G$
- $b'$ random, $q$ curve order
ECC SCA threats big picture

EC{*}DSA signatures
Message \( m \), Nonce \( k \), priv key \( x \)
\[ W = k \times G = (W_x, W_y) \]
\[ r = W_x \parallel s = k^{-1}(x \times r + H(m)) \]

EC scalar multiplication
\[ kP = k \times P \]
\( k = \{k_0, k_1, \ldots, k_n\} \Rightarrow \text{DBL/ADD} \)

EC Double and Add
\[ \text{ADD}(P, Q) / \text{DBL}(P) \]
Point at infinity \( O \) handling
Affine/Projective coordinates

N and \( \mathbb{F}_p \)
Big number operations
Size trimming
Division, inversion

- Find one \( k \) with SPA ⇒ recover \( x \)
- Collect multiple \( k_i \)
  - MSB timing
  - Use Hidden Number Problem ⇒ recover \( x \)
  (2019/2020)

Use scalar mult

Single trace
Multiple traces

Double and Add variants

Double/Add formulas

- Distinguish \( \text{ADD} \) and \( \text{DBL} \) (non-unified formulas)
- Distinguish point at infinity \( O \) (incomplete formulas)

- Non-constant time operations (extra reductions, …)
- Operands address/value usage collisions (horizontal attacks)

- Arithmetic blinding on \( k \) and \( x \):
  \[ (b \times k) - 1 (x \times r + b \times H(m)) \]

- Scalar multiplication blinding on \( k \):
  \[ k \times G = (k + b' \times q) \times G \]
  \( b' \) random, \( q \) curve order
### ECC SCA Threats Big Picture

**EC(*)DSA signatures**
- Message $m$, Nonce $k$, priv key $x$
  - $W = k \times G = (W_x, W_y)$
  - $r = W_x || s = k^{-1}(x \times r + H(m))$

**EC scalar multiplication**
- $kP = k \times P$
- $k = \{k_0, k_1, \ldots, k_n\} \Rightarrow$ DBL/ADD

**EC Double and Add**
- ADD($P, Q$) / DBL($P$)
- Point at infinity $O$ handling
- Affine/Projective coordinates

**N and $\mathbb{F}_p$**
- Big number operations
  - Size trimming
  - Division, inversion

#### Attack double and add variants with SPA, DPA, horizontal attacks ⇒ recover $k$
- Single or multiple traces

- Non-constant time operations (extra reductions, …)
- Operands address/value usage collisions (horizontal attacks)

- Arithmetic blinding on $k$ and $x$:
  - $(b \times k)^{-1}(x \times b \times r + b \times H(m))$

- Scalar multiplication blinding on $k$:
  - $k \times G = (k + b' \times q) \times G$
    - $b'$ random, $q$ curve order
**ECC SCA Threats Big Picture**

**EC(*)DSA signatures**
Message $m$, Nonce $k$, priv key $x$
$W = k \times G = (W_x, W_y)$
$r = W_x \parallel s = k^{-1}(x \times r + H(m))$

Use scalar mult

**EC scalar multiplication**
$kP = k \times P$
$k = \{k_0, k_1, ..., k_n\} \Rightarrow DBL/ADD$

**EC Double and Add**
$ADD(P, Q) / DBL(P)$
Point at infinity $O$ handling
Affine/Projective coordinates

- Distinguish $ADD$ and $DBL$
  (non-unified formulas)
- Distinguish point at infinity $O$
  (incomplete formulas)
...

Double/Add formulas

**N and $\mathbb{F}_p$**
Big number operations
Size trimming
Division, inversion

Single or multiple traces
ECC SCA threats big picture

**EC(*)DSA signatures**
Message m, Nonce k, priv key x
W = k × G = (Wx, Wy)
r = Wx||s = k⁻¹(x × r + H(m))

**Use scalar mult**

**EC scalar multiplication**
kP = k × P
k = \{k₀, k₁, ..., kₙ\} ⇒ DBL/ADD

**Double and Add variants**

**EC Double and Add**

**ADD(P, Q) / DBL(P)**
Point at infinity O handling
Affine/Projective coordinates

**Double/Add formulas**

**N and \( \mathbb{F}_p \)**

Big number operations
Size trimming
Division, inversion

- Non-constant time operations
  (extra reductions, ...)
- Operands address/value usage collisions
  (horizontal attacks)

Single or multiple traces
### libecc SCA Countermeasures

#### EC(*)DSA signatures
- Find one $k$ with SPA $\Rightarrow$ recover $x$
- Collect partial information on multiple $k_i$ (MSB timing, SPA, ...)
- Use Hidden Number Problem $\Rightarrow$ recover $x$

#### EC scalar multiplication
- Attack double and add variants with SPA, DPA, horizontal attacks $\Rightarrow$ recover $k$

#### EC Double and Add
- Distinguish ADD and DBL (non-unified formulas)
- Distinguish point at infinity $\mathcal{O}$ (incomplete formulas)

#### Big number operations
- Non-constant time operations (extra reductions, ...)
- Operands address/value usage collisions (horizontal attacks)

**Libecc SCA Countermeasures**

**Message $m$, Nonce $k$, priv key $x$**

$W = k \times G = (W_x, W_y)$

$r = W_x || s = k^{-1}(x \times r + H(m))$

**Use scalar mult**

**kP = k \times P**

$k = \{k_0, k_1, ..., k_n\} \Rightarrow$ DBL/ADD

**Double and Add variants**

**ADD(P, Q) / DBL(P)**

Point at infinity $\mathcal{O}$ handling

Affine/Projective coordinates

**Double/Add formulas**

**N and $\mathbb{F}_p$**

Big number operations

Size trimming

Division, inversion

**Forced by user (huge performance hit)**
# libecc SCA countermeasures

## EC{*}DSA signatures
- Message \( m \), Nonce \( k \), priv key \( x \)
  \[
  W = k \times G = (W_x, W_y) \\
  r = W_x || s = k^{-1}(x \times r + H(m))
  \]

## EC scalar multiplication
- \( kP = k \times P \)
  \( k = \{k_0, k_1, \ldots, k_n\} \Rightarrow \text{DBL/ADD} \)

## EC Double and Add
- \( \text{ADD}(P, Q) / \text{DBL}(P) \)
  - Point at infinity \( O \) handling
  - Affine/Projective coordinates

## \( \mathbb{F}_p \)
- Big number operations
  - Size trimming
  - Division, inversion

## SCA countermeasures
- Arithmetic blinding on \( k \) and \( x \):
  \[
  (b \times k)^{-1}(x \times b \times r + b \times H(m))
  \]
- Scalar multiplication blinding on \( k \):
  \[
  k \times G = (k + b' \times q) \times G \\
  (b' \text{ random, } q \text{ curve order})
  \]

## Attack double and add variants with SPA, DPA, horizontal attacks ⇒ recover \( k \)

## Double/Add formulas
- Distinguish \( \text{ADD} \) and \( \text{DBL} \)
  (non-unified formulas)
- Distinguish point at infinity \( O \)
  (incomplete formulas)

## Non-constant time operations
- (extra reductions, ...)
- Operands address/value usage collisions
  (horizontal attacks)

## Distinguish ADD and DBL formulas
- USE Hidden Number Problem ⇒ recover \( x \)

## Find one \( k \) with SPA
- Collect partial information on multiple \( k_i \)
  (MSB timing, SPA, ...)
- Use Hidden Number Problem ⇒ recover \( x \)

## Attack double and add variants with SPA, DPA, horizontal attacks
- Distinguish \( \text{ADD} \) and \( \text{DBL} \)
- Distinguish point at infinity \( O \)
- Non-constant time operations
- Operands address/value usage collisions
- Operands address/value usage collisions

## Arithmetic blinding on \( k \) and \( x \):
- Find one \( k \) with SPA
- Collect partial information on multiple \( k_i \)
- Use Hidden Number Problem ⇒ recover \( x \)

## Non-constant time operations
- Operands address/value usage collisions
  (horizontal attacks)
**libecc SCA COUNTERMEASURES**

**EC(*)DSA signatures**
Message $m$, Nonce $k$, priv key $x$

$$W = k \times G = (W_x, W_y)$$

$$r = W_x \| s = k^{-1}(x \times r + H(m))$$

- Use scalar mult
- Arithmetic blinding on $k$ and $x$: $$(b \times k)^{-1}(x \times b \times r + b \times H(m))$$
- Scalar multiplication blinding on $k$:
  $$k \times G = (k + b' \times q) \times G$$
  ($b'$ random, $q$ curve order)

**EC scalar multiplication**

$$kP = k \times P$$

$$k = \{k_0, k_1, \ldots, k_n\} \Rightarrow$$ DBL/ADD

- Optional scalar blinding + force MSB (small order multiple)
- Montgomery Ladder or Double-and-Add-Always
- projective coordinates masking DDPA, Itoh et al. masking ADPA

**EC Double and Add**

$$\text{ADD}(P, Q) / \text{DBL}(P)$$

Point at infinity $O$ handling
Affine/Projective coordinates

- Distinguish ADD and DBL (non-unified formulas)
- Distinguish point at infinity $O$ (incomplete formulas)
- ... 

**N and $\mathbb{F}_p$**

Big number operations
Size trimming
Division, inversion

- Non-constant time operations (extra reductions, ...)
- Operands address/value usage collisions (horizontal attacks)
- ...
**libecc SCA countermeasures**

- **EC{*}DSA signatures**
  - Message $m$, Nonce $k$, priv key $x$
  - $W = k \times G = (W_x, W_y)$
  - $r = W_x || s = k^{-1}(x \times r + H(m))$

- **EC scalar multiplication**
  - $kP = k \times P$
  - $k = \{k_0, k_1, ..., k_n\} \Rightarrow$ DBL/ADD

- **EC Double and Add**
  - $\text{ADD}(P, Q) / \text{DBL}(P)$
  - Point at infinity $O$ handling
  - Affine/Projective coordinates

- **N and $\mathbb{F}_p$**
  - Big number operations
  - Size trimming
  - Division, inversion

- **Use scalar mult**

  - Arithmetic blinding on $k$ and $x$:
    
    $$(b \times k)^{-1}(x \times b \times r + b \times H(m))$$

  - Scalar multiplication blinding on $k$:
    
    $$k \times G = (k + b' \times q) \times G$$
    
    ($b'$ random, q curve order)

- **Double and Add variants**

  - Optional scalar blinding + force MSB (small order multiple)
  - Montgomery Ladder or Double-and-Add-Always
  - projective coordinates masking DDPA, Itoh et al. masking ADPA

- **Double/Add formulas**

  - Use Montgomery Ladder or Double-and-Add-Always
  - Use complete formulas
  - Limit leaking $O$ when checking it

- **- Non-constant time operations**
  - (extra reductions, ...)
  - Operands address/value usage collisions
  - (horizontal attacks)
  - ...

- **Optionalscalar blinding + force MSB**

  - Random, q curve order

- **Optionalscalar blinding + force MSB**

  - Random, q curve order

- **Optionalscalar blinding + force MSB**

  - Random, q curve order

- **Optionalscalar blinding + force MSB**

  - Random, q curve order

- **Optionalscalar blinding + force MSB**

  - Random, q curve order

- **Optionalscalar blinding + force MSB**

  - Random, q curve order
**libecc SCA Countermeasures**

- **EC(*)DSA signatures**
  - Message $m$,Nonce $k$, priv key $x$
  - $W = k \times G = (W_x, W_y)$
  - $r = W_x \parallel s = k^{-1}(x \times r + H(m))$

- **Use scalar mult**

- **EC scalar multiplication**
  - $kP = k \times P$
  - $k = \{k_0, k_1, \ldots, k_n\} \Rightarrow \text{DBL/ADD}$

- **Double and Add variants**

- **EC Double and Add**
  - $\text{ADD}(P, Q) / \text{DBL}(P)$
  - Point at infinity $O$ handling
  - Affine/Projective coordinates

- **Double/Add formulas**

- **N and $\mathbb{F}_p$**
  - Big number operations
  - Size trimming
  - Division, inversion

- **- Arithmetic blinding on $k$ and $x$:**
  - $(b \times k)^{-1}(x \times b \times r + b \times H(m))$
- **- Scalar multiplication blinding on $k$:**
  - $k \times G = (k + b' \times q) \times G$
  - ($b'$ random, q curve order)

- **Optional scalar blinding + force MSB** (small order multiple)
- **- Montgomery Ladder or Double-and-Add-Always**
- **- projective coordinates masking DDPA, Itoh et al. masking ADPA**

- **- Use Montgomery Ladder or Double-and-Add-Always**
- **- Use complete formulas**
- **- Limit leaking $O$ when checking it**

- **- Try to achieve constant time where possible**
- **- Use operands bits independent branches and choices**
- **- Offer or ensure non-trimming primitives**...
**libecc SCA COUNTERMEASURES**

- **EC(*)DSA signatures**
  - Message $m$, Nonce $k$, priv key $x$
  - $W = k \times G = (W_x, W_y)$
  - $r = W_x || s = k^{-1}(x \times r + H(m))$

- **EC scalar multiplication**
  - $kP = k \times P$
  - $k = \{k_0, k_1, \ldots, k_n\} \Rightarrow$ DBL/ADD

- **EC Double and Add**
  - $\text{ADD}(P, Q) / \text{DBL}(P)$
  - Point at infinity $O$ handling
  - Affine/Projective coordinates

- **N and $\mathbb{F}_p$**
  - Big number operations
  - Size trimming
  - Division, inversion

- **Use scalar mult**

- **EC Double and Add variants**
  - Double and Add formulas

- **Use complete formulas**
  - Limit leaking $O$ when checking it

- **- Optional scalar blinding + force MSB** (small order multiple)
  - - Montgomery Ladder or Double-and-Add-Always
  - - projective coordinates masking DDPA, Itoh et al. masking ADPA

- **- Try to achieve constant time** where possible
  - - Use operands bits independent branches and choices
  - - Offer or ensure non-trimming primitives ...

- **- Arithmetic blinding on $k$ and $x$**:
  - $(b \times k)^{-1}(x \times b \times r + b \times H(m))$

- **- Scalar multiplication blinding on $k$**:
  - $k \times G = (k + b' \times q) \times G$
  - $(b'$ random, $q$ curve order)

- **Forced by user**
  - (huge performance hit)

- **- Find one $k$ with SPA**
  - - recover $x$

- **- Collect partial information on multiple $k_i$ (MSB timing, SPA, …)**

- **- Use Hidden Number Problem**
  - - recover $x$

- **- Attack double and add variants with SPA, DPA, horizontal attacks**
  - - recover $k$

- **- Distinguish $\text{ADD}$ and $\text{DBL}$ (non-unified formulas)**

- **- Distinguish point at infinity $O$ (incomplete formulas)**

- **- Non-constant time operations** (extra reductions, …)

- **- Operands address/value usage collisions** (horizontal attacks) ...

- **- Arithmetic blinding on $k$ and $x$:**
  - $(b \times k)^{-1}(x \times b \times r + b \times H(m))$

- **- Scalar multiplication blinding on $k$:**
  - $k \times G = (k + b' \times q) \times G$
  - $(b'$ random, $q$ curve order)
**Focus on constant time/trace**

- **Constant time** is very hard to achieve:
  - Low level issues on heterogeneous platforms
    - Quirks on some instructions or accelerated IPs, ...
  - Compilation from C code
    - Portability versus constant time dilemma
Focus on constant time/trace

- **Constant time** is very hard to achieve:
  - Low level issues on heterogeneous platforms
  - Quirks on some instructions or accelerated IPs, ...
  - Compilation from C code
  - Portability versus constant time dilemma

- **Constant trace** is even harder to achieve:
  - Heavily depends on the platform
  - New microarchitectural SCA (speculative execution and so on)
Focus on constant time/trace

- **Constant time** is very hard to achieve:
  - Low level issues on heterogeneous platforms
  - Quirks on some instructions or accelerated IPs, ...
  - Compilation from C code
  - Portability versus constant time dilemma

- **Constant trace** is even harder to achieve:
  - Heavily depends on the platform
  - New microarchitectural SCA (speculative execution and so on)

**Fine-grained local** SCA resistance is probably a *lost battle* (at least in C)
Focus on constant time/trace

Constant time is very hard to achieve:

- Low level issues on heterogeneous platforms
  ⇒ Quirks on some instructions or accelerated IPs, ...
- Compilation from C code
  ⇒ Portability versus constant time dilemma

Constant trace is even harder to achieve:

- Heavily depends on the platform
- New microarchitectural SCA (speculative execution and so on)

Fine-grained local SCA resistance is probably a lost battle
(at least in C)

Characterization on a specific target is always necessary!
libecc is (and will) never be advertised as SCA resistant
libecc SCA countermeasures wrap-up

- libecc should be hopefully:
  - Remote timing-attack resistant
  - SPA resistant
  - Data-DPA and Address-DPA resistant
libecc SCA countermeasures wrap-up

- libecc should be hopefully:
  - Remote timing-attack resistant
  - SPA resistant
  - Data-DPA and Address-DPA resistant

- libecc might (probably) not be:
  - Fully horizontal attacks resistant
**libecc SCA countermeasures wrap-up**

- **libecc** should be **hopefully**:
  - Remote timing-attack resistant
  - SPA resistant
  - Data-DPA and Address-DPA resistant

- **libecc** might (probably) not be:
  - Fully horizontal attacks resistant

**Known future work** to increase resistance:
- Constant time division and inversion
- Check point at infinity $\infty$ handling trace
- Improve dummy operations stealthiness
- ...

---

**libecc is a work in progress**
(Open-source community)
A long history of fault attacks on ECC:

- Invalid points injections and analysis
- Invalid and twisted curves analysis
- Fine-grained faults models
- ...

libecc has put minimal efforts to thwart faults:

- Complete formulas to properly handle
- Point on curve checks when importing/exporting a point
- Affine to projective, projective to affine transformations
- Point on curve checks when entering and leaving scalar multiplication
- No robust if and loops, nor robust return values, ...

⇒ future work
A long history of fault attacks on ECC:
- Invalid points injections and analysis
- Invalid and twisted curves analysis
- Fine-grained faults models
- ...

libecc has put minimal efforts to thwart faults:
- Complete formulas to properly handle \( O \)
- Point on curve checks when importing/exporting a point
  \( \Rightarrow \) Affine to projective, projective to affine transformations
- Point on curve checks when entering and leaving scalar multiplication
**LIBECC AND FAULT ATTACKS**

- A long history of fault attacks on ECC:
  - Invalid points injections and analysis
  - Invalid and twisted curves analysis
  - Fine-grained faults models
  - ...

- libecc has put **minimal efforts** to thwart faults:
  - Complete formulas to properly handle $O$
  - Point on curve checks when importing/exporting a point
    $\Rightarrow$ Affine to projective, projective to affine transformations
  - Point on curve checks when entering and leaving scalar multiplication
  - No robust if and loops, nor robust return values, ... $\Rightarrow$ future work

**Characterization** on a specific target is **always** necessary!
(glitch sensitivity, glitch captors robustness, ...)
libecc usecases: some examples
external token (ST33)

ST33

SoC

RESET

I²C SDA

I²C SCL

MISO

MOSI

CS#

CLK

libecc - SemSecuElec February 2020 - Rennes
external token
(ST33)

I²C SDA

I²C SCL

ST33

SoC

RESET

MISO
MOSI
CS#
CLK

libecc - SemSecuElec february 2020 - Rennes
Secure channel establishment

**Mutual authentication** over I^2^C and SPI
**Secure channel establishment**

- Mutual authentication over I²C and SPI
- Factorized code using **libec** (ECDH, ECDSA)

External token (ST33)

Authenticated channel

**libec**

Secure channel establishment

Mutual authentication over I²C and SPI

Factorized code using **libec** (ECDH, ECDSA)

External token (ST33)

Authenticated channel

**libec**

Secure channel establishment

Mutual authentication over I²C and SPI

Factorized code using **libec** (ECDH, ECDSA)

External token (ST33)

Authenticated channel

**libec**

Secure channel establishment

Mutual authentication over I²C and SPI

Factorized code using **libec** (ECDH, ECDSA)

External token (ST33)

Authenticated channel

**libec**

Secure channel establishment

Mutual authentication over I²C and SPI

Factorized code using **libec** (ECDH, ECDSA)

External token (ST33)

Authenticated channel
The WooKey project
The WooKey project

libecc on STM32F439

Secure channel

Javacard
Conclusion

Open-source ECC library:

- Aiming at simplicity and portability
- Code robustness
- Some protections against SCA ...
- ... very few against faults
Conclusion

Open-source ECC library:

- Aiming at simplicity and portability
- Code robustness
- Some protections against SCA ...
- ... very few against faults

Future work:

- Continue SCA resistance improvements
  ⇒ Algorithmic and through characterization
- Integrate fault resistant primitives
- Add RSA algorithm and plug with Frama-C X.509 parser
- ...any interesting ideas!