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

Layout of the presentation

- 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-2019-1547 (OpenSSL advisory) [Low severity] 10 September 2019:

> Normally in OpenSSL EC groups always have a co-factor present and this is used in side channel resistant code paths. However, in some cases, it is possible to construct a group using explicit parameters (instead of using a named curve). In those cases it is possible that such a group does not have the cofactor present. This can occur even where all the parameters match a known named curve. If such a curve is used then OpenSSL falls back to non-side channel resistant code paths which may result in full key recovery during an ECDSA signature operation. In order to be vulnerable an attacker would have to have the ability to time the creation of a large number of signatures where explicit parameters with no co-factor present are in use by an application using libcrypto. For the avoidance of doubt libssl is not vulnerable because explicit parameters are never used. Reported by Cesar Pereida García, Sohaib ul Hassan, Nicola Tuveri, Iaroslav Gridin, Alejandro Cabrera Aldaya, and Billy Brumley.
>
> - Fixed in OpenSSL 1.1.1d (git commit) (Affected 1.1.1-1.1.1c)
> - Fixed in OpenSSL 1.1.0l (git commit) (Affected 1.1.0-1.1.0k)
> - Fixed in OpenSSL 1.0.2t (git commit) (Affected 1.0.2-1.0.2s)

Title	**Side channel attack on ECDSA**
CVE	CVE-2019-18222
Date	15th January 2020
Affects	All versions of Mbed TLS and Mbed Crypto
Impact	The private key is recoverable through side channels.
Severity	High
Credit	Alejandro Cabrera Aldaya and Billy Brumley

Vulnerability

Our bignum implementation is not constant time/constant trace, so side channel attacks can retrieve the blinded value, factor it (as it is smaller than RSA keys and not guaranteed to have only large prime factors), and then, by brute force, recover the key. Reported by Alejandro Cabrera Aldaya and Billy Brumley.

Impact

If the adversary is in the position to launch a cache attack, then they may be able to recover the private key.

Resolution

Affected users should upgrade to one of the most recent versions of Mbed TLS, including 2.20.0, 2.16.4 or 2.7.13 or later. Similarly, affected users should upgrade to the most recent version of Mbed Crypto, including 3.0.1 or later.

edit: Earlier, this document falsely stated that Mbed Crypto 3.0.0 fixes this issue. This is not true, Mbed Crypto 3.0.1 is the earliest version with the fix. Users should upgrade to Mbed Crypto 3.0.1 or later.

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-2019-1547 (OpenSSL advisory) [Low severity] 10 September 2019:

Normally in OpenSSL EC groups always have a co-factor present and this is used in side channel resistant code paths. However, in some cases, it is possible to construct a group using explicit parameters (instead of using a named curve). In those cases it is possible that such

CVE-2018-0739 (OpenSSL advisory) [Moderate severity] 27 March 2018:

Constructed ASN.1 types with a recursive definition (such as can be found in PKCS7) could eventually exceed the stack given malicious input with excessive recursion. This could result in a Denial Of Service attack. There are no such structures used within SSL/TLS that come from untrusted sources so this is considered safe. Reported by OSS-fuzz.

- Fixed in OpenSSL 1.1.0h (git commit) (Affected 1.1.0-1.1.0g)
- Fixed in OpenSSL 1.0.2o (git commit) (Affected 1.0.2b-1.0.2n)

Alejandro Cabrera Aldaya, and Billy Brumley.

- Fixed in OpenSSL 1.1.1d (git commit) (Affected 1.1.1-1.1.1c)
- Fixed in OpenSSL 1.1.0l (git commit) (Affected 1.1.0-1.1.0k)
- Fixed in OpenSSL 1.0.2t (git commit) (Affected 1.0.2-1.0.2s)

Title	Side channel attack on ECDSA
CVE	CVE-2019-18222
Date	15th January 2020
Affects	All versions of Mbed TLS and Mbed Crypto
Impact	The private key is recoverable through side channels.

Title	**Risk of remote code execution when truncated HMAC is enabled**
CVE	CVE-2018-0488
Date	1st February 2018
Affects	All versions of Mbed TLS from version 1.3.0 and up, including all 2.1 and later releases
Not affected	Mbed TLS 1.2.19 and earlier
Impact	Enabling truncated HMAC can lead to remote code execution
Severity	High

Resolution

Affected users should upgrade to one of the most recent versions of Mbed TLS, including 2.20.0, 2.16.4 or 2.7.13 or later. Similarly, affected users should upgrade to the most recent version of Mbed Crypto, including 3.0.1 or later.

edit: Earlier, this document falsely stated that Mbed Crypto 3.0.0 fixes this issue. This is not true. Mbed Crypto 3.0.1 is the earliest version with the fix. Users should upgrade to Mbed Crypto 3.0.1 or later.

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 $\mathbb{N}$
  - 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`
  (dual BSD / GPLv2+ license)

Some limitations

- Only curves over $\mathbb{F}_p$ in the Weierstraß form: $\boxed{y^2 = x^3 + a \times x + b}$
- 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: $y^2 = x^3 + a \times x + b$
- Means:
  - No support of curves over binary fields $\mathbb{F}_{2^m}$
  - No Edwards curves (i.e. no Ed25519 and so on)
- S… curves)
- P…
  - More on this later

Frama-C validated `X.509` parser,
free of RunTime Errors:
`https://github.com/ANSSI-FR/x509-parser`

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

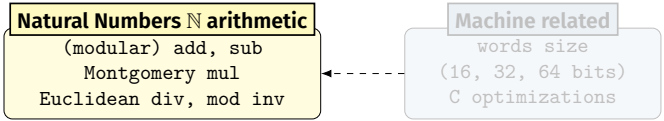
Library internals: *architecture* and *code*

LIBECC ARCHITECTURE

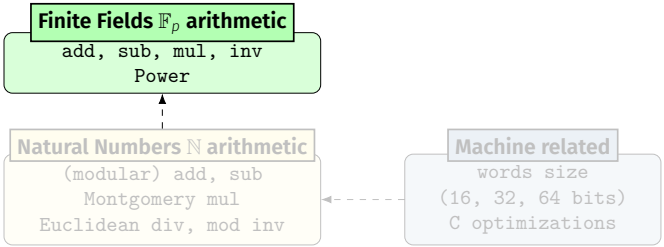
Machine related

words size
(16, 32, 64 bits)
C optimizations

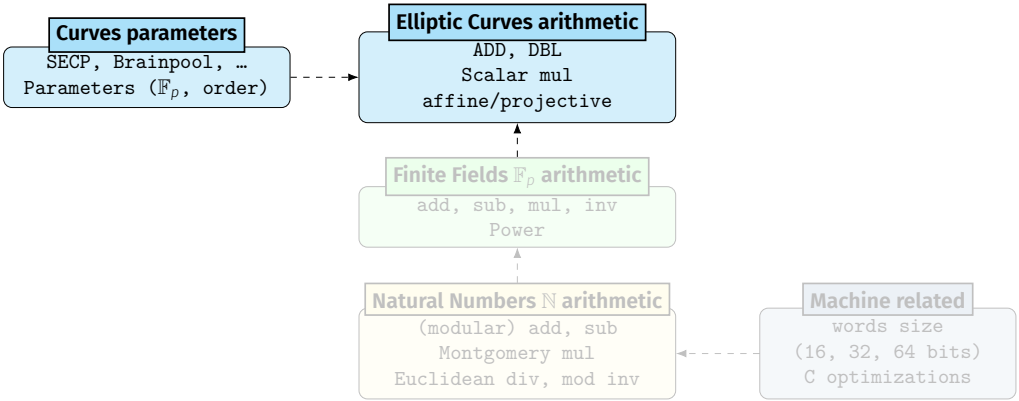
LIBECC ARCHITECTURE



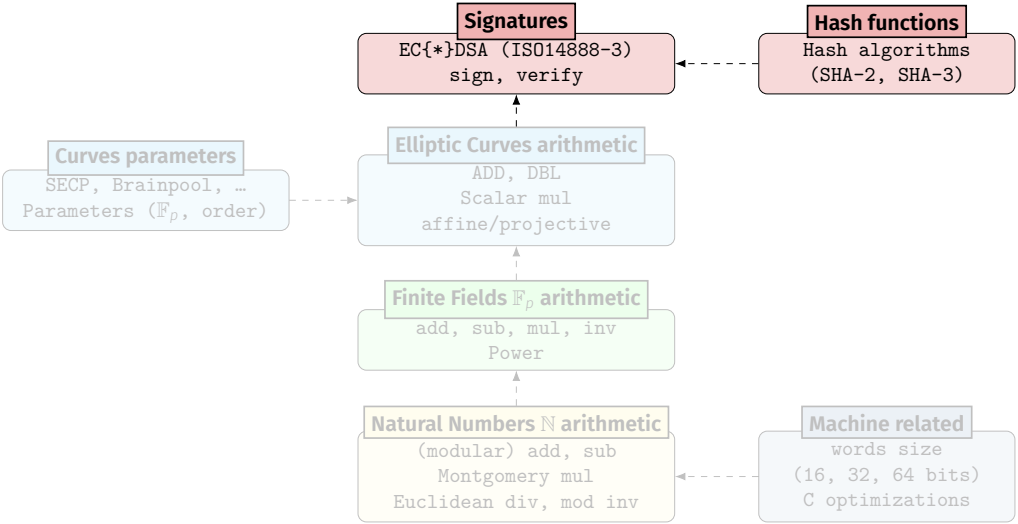
LIBECC ARCHITECTURE



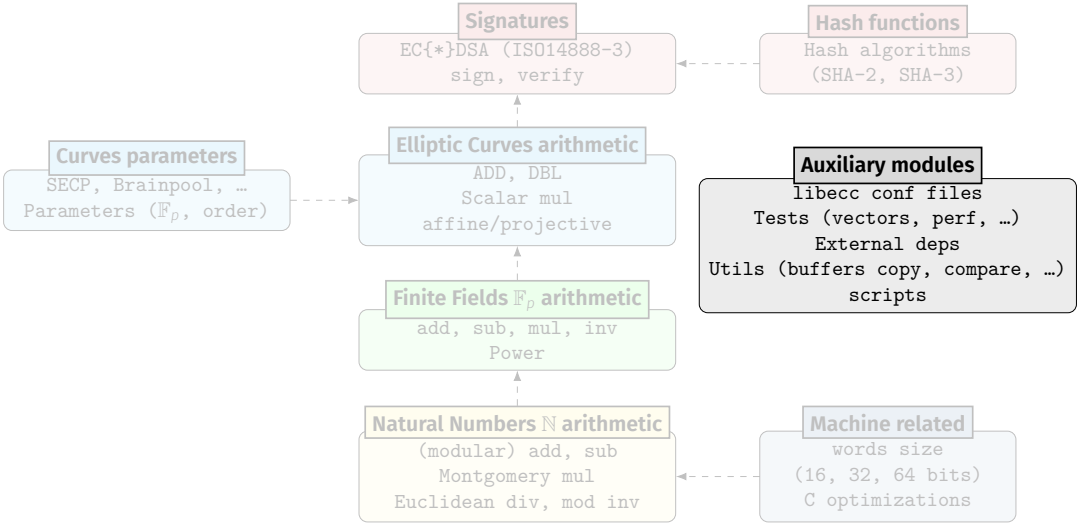
LIBECC ARCHITECTURE



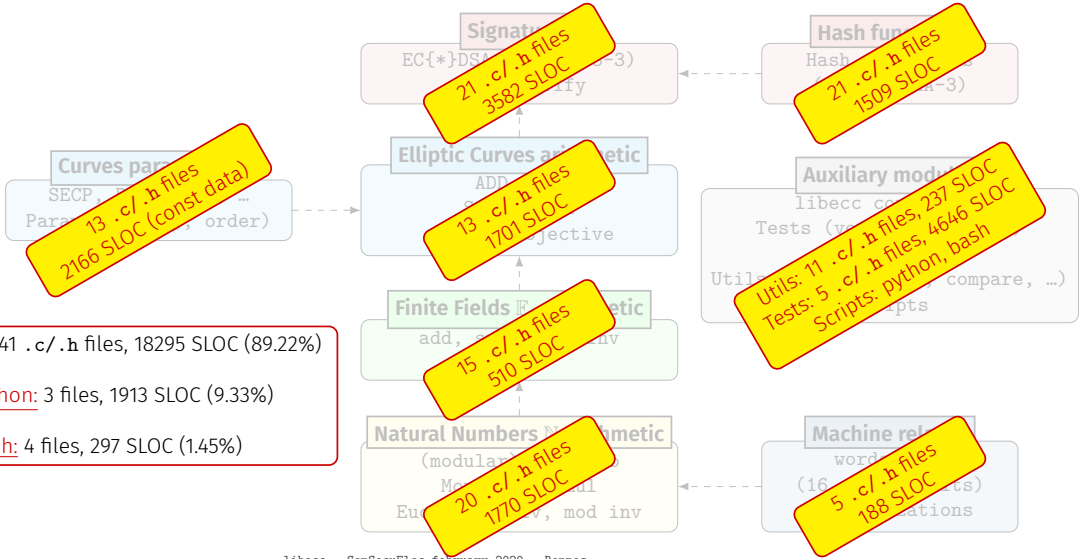
LIBECC ARCHITECTURE



LIBECC ARCHITECTURE



LIBECC ARCHITECTURE



C: 141 `.c/.h` files, 18295 SLOC (89.22%)

Python: 3 files, 1913 SLOC (9.33%)

Bash: 4 files, 297 SLOC (1.45%)

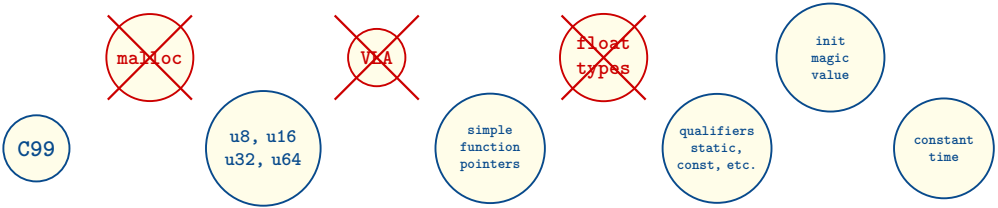
LIBECC CONFIGURATION FILE

```
$ cat lib_ecc_config.h
```

```c
#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
#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
/* Supported sig/verif schemes */
#define WITH_SIG_ECDSA
#define WITH_SIG_ECKCDSA
#define WITH_SIG_ECSDSA
#define WITH_SIG_ECOSDSA
#define WITH_SIG_ECFSDSA
#define WITH_SIG_ECGDSA
#define WITH_SIG_ECRDSA

#endif /* __LIB_ECC_CONFIG_H__ */
```

Coding style and rationale

- 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 suport
  - 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 suport
  - 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="-mgbz80 --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
...
```

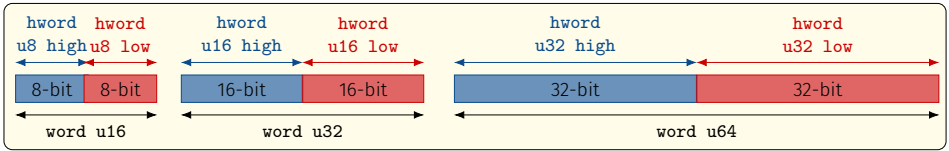
Tests

- 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)
  - Cross-build (`docker`) and cross-run (`qemu-static`) scripts
    $\Rightarrow$ Extensive $\approx$ 200 `travis` jobs matrix for non-regression

✓	# 6.16	AMD64	C	CC=clang BLINDING=1 COMPLETE=1 CROSS_BUILD=0 LADDER=0		13 min 27 sec
✓	# 6.17	AMD64	C	BLINDING=0 COMPLETE=0 CROSS_BUILD=1 CROSS_TARGET=arm-linux-gnueabi C		23 min 42 sec
✓	# 6.18	AMD64	C	BLINDING=1 COMPLETE=0 CROSS_BUILD=1 CROSS_TARGET=arm-linux-gnueabi C		27 min 32 sec
✓	# 6.19	AMD64	C	BLINDING=0 COMPLETE=1 CROSS_BUILD=1 CROSS_TARGET=arm-linux-gnueabi C		26 min 9 sec
✓	# 6.20	AMD64	C	BLINDING=1 COMPLETE=1 CROSS_BUILD=1 CROSS_TARGET=arm-linux-gnueabi C		30 min 35 sec

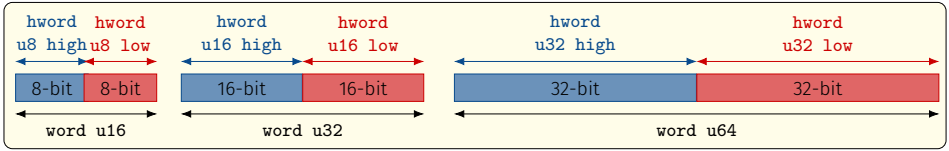
Big numbers representation

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

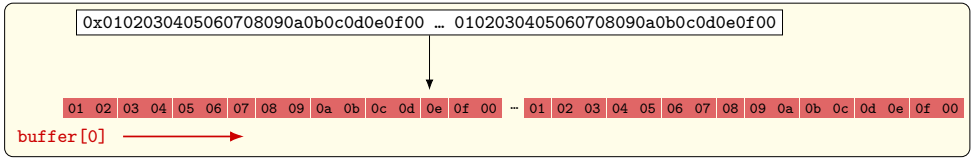


Big numbers representation

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

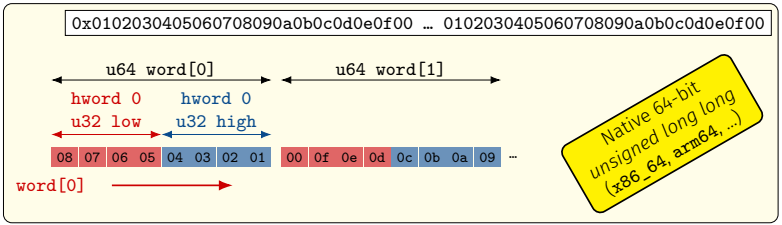


  - Input/output representation is classical big endian



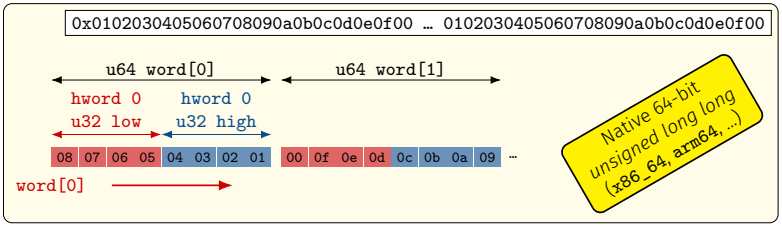
Big numbers representation

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



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

External dependencies

- The core library has no dependency, not even libc
- External dependencies:
  Critical:
  - Random source

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

External dependencies

- 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)

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

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

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, ...

```
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

"Fair" comparison (no NIST optimizations)

Comparison for BrainpoolP256r1			
library	sign	verify	RAM/ROM
Core i7-5500U @ 2.40GHz			
libecc	500/s	250/s	-
mbedTLS	426/s	106/s	-
OpenSSL	2463/s	1767/s	-
Marvell Armada A388 @ 1.60GHz			
libecc	55/s	28/s	-
mbedTLS	33/s	8/s	-
OpenSSL	369/s	332/s	-
Cortex-M3 MCU @ 72MHz			
libecc	0.91/s	0.45/s	8KB/32KB
mbedTLS	0.42/s	0.21/s	3KB/32KB
OpenSSL	NA	NA	NA

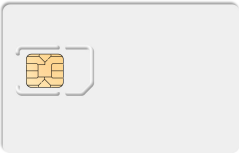
High end

Embedded

Very constrained MCU

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 */
    ...
}
```

```c
/* 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 */
    ...
}
```

```c
/* 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 */
    ...
}
```

```c
/* 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 */
    ...
}
```

```c
/* 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 */
    ...
}
```

```c
/* 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

```
/* 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
{
    MUST_HAVE ((A != NULL) && (A->magic == MA
}

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                    */
    math_object
    ...
    /* Ch                    tion */
    ma                    alized(in1);
                          initialized(in2);
                  sing, initialize output */
                  = in1) && (out != in2)) {
                  math_object_init(out, ...);
                  ...
    }
    /* Algorithm core */
    ...
    /* Uninitialize local variables */
    math_object_uninit(&tmp);
    return;
}
```

Static analysis
whenever/wherever possible!

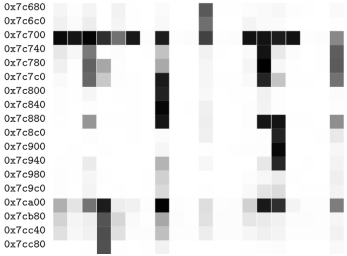
ECC PASSIVE PHYSICAL ATTACKS

- 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, ...



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)

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

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

$kP = k \times P$

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

Double and Add variants

EC Double and Add

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

Point at infinity $\mathcal{O}$ 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

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

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

Use scalar mult

- Find one k with SPA $\Rightarrow$ recover x — Single trace

- Collect partial information on multiple k_i (MSB timing, SPA, ...)
- Use Hidden Number Problem $\Rightarrow$ recover x — Multiple traces

EC scalar multiplication

$kP = k \times P$

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

Double and Add variants

EC Double and Add

ADD(P, Q) / DBL(P)

Point at infinity $\mathcal{O}$ 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

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

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

Use scalar mult

- Find one k with SPA $\Rightarrow$ recover x — Single trace

- Collect … tiple k_i
(MSB tim …
- Use Hid … cover x — Multiple traces

MSB timing
$\Rightarrow$ Minerva and TPM.fail
(2019/2020)

EC scalar multiplication

$kP = k \times P$

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

Double and Add variants

EC Double and Add

ADD(P, Q) / DBL(P)

Point at infinity $\mathcal{O}$ 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

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

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

Use scalar mult

EC scalar multiplication

$kP = k \times P$

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

Double and Add variants

Attack double and add variants with SPA, DPA, horizontal attacks $\Rightarrow$ recover k

Single or multiple traces

EC Double and Add

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

Point at infinity $\mathcal{O}$ 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

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

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

Use scalar mult

EC scalar multiplication

$kP = k \times P$

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

Double and Add variants

EC Double and Add

ADD(P, Q) / DBL(P)

Point at infinity $\mathcal{O}$ handling

Affine/Projective coordinates

Double/Add formulas

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

...

Single or multiple traces

$\mathbb{N}$ 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

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

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

Use scalar mult

EC scalar multiplication

$kP = k \times P$

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

Double and Add variants

EC Double and Add

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

Point at infinity $\mathcal{O}$ handling

Affine/Projective coordinates

Double/Add formulas

$\mathbb{N}$ 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

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

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

EC scalar multiplication

$kP = k \times P$

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

Double and Add variants

Attack double and add variants with SPA, DPA, horizontal attacks ⇒ recover k

EC Double and Add

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

Point at infinity $\mathcal{O}$ handling

Affine/Projective coordinates

Double/Add formulas

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

...

$\mathbb{N}$ 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

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

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

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

$kP = k \times P$

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

Double and Add variants

Attack double and add variants with SPA, DPA, horizontal attacks $\Rightarrow$ recover k

EC Double and Add

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

Point at infinity $\mathcal{O}$ handling

Affine/Projective coordinates

Double/Add formulas

- Distinguish `ADD` and `DBL`
(non-unified formulas)
- Distinguish point at infinity $\mathcal{O}$
(incomplete formulas)
...

$\mathbb{N}$ 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

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

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

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

$kP = k \times P$

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

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

EC Double and Add

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

Point at infinity $\mathcal{O}$ handling

Affine/Projective coordinates

Double/Add formulas

- Distinguish `ADD` and `DBL`
(non-unified formulas)
- Distinguish point at infinity $\mathcal{O}$
(incomplete formulas)
...

$\mathbb{N}$ 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

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

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

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

$kP = k \times P$

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

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

EC Double and Add

ADD(P, Q) / DBL(P)

Point at infinity $\mathcal{O}$ handling

Affine/Projective coordinates

Double/Add formulas

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

$\mathbb{N}$ 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

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

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

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

$kP = k \times P$

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

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

EC Double and Add

ADD(P, Q) / DBL(P)

Point at infinity $\mathcal{O}$ handling

Affine/Projective coordinates

Double/Add formulas

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

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

Big number operations

Size trimming

Division, inversion

- 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

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

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

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)

Forced by user
(huge performance hit)

EC scalar multiplication

$kP = k \times P$

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

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

EC Double and Add

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

Point at infinity $\mathcal{O}$ handling

Affine/Projective coordinates

Double/Add formulas

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

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

Big number operations

Size trimming

Division, inversion

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

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 advertized 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 inifinity $\mathcal{O}$ handling trace
  - Improve dummy operations stealthiness
  - ...

libecc is a **work in progress**
(Open-source community)

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 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 $\mathcal{O}$
  - 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

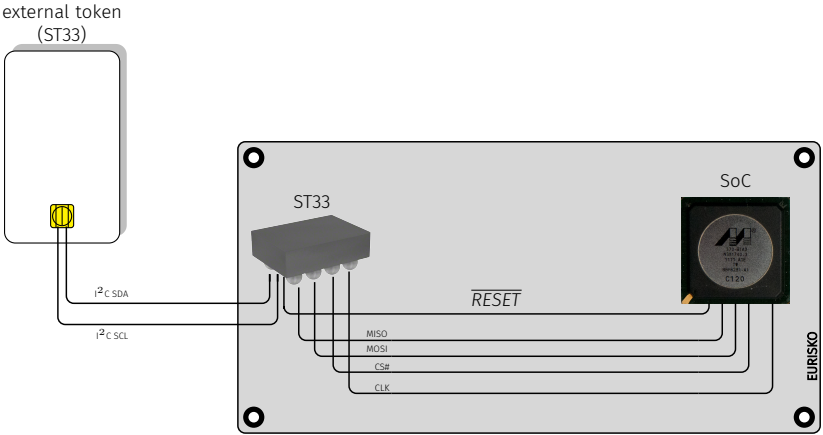
- A long history of fault attacks on ECC :
  - Invalid points injections and analysis
  - Invalid and twisted curves analysis
  - Fine-grained faults models
  - ...
- libecc has put minimal efforts to thwart faults:
  - Complete formulas to properly handle $\mathcal{O}$
  - Point on curve checks when 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, ...)

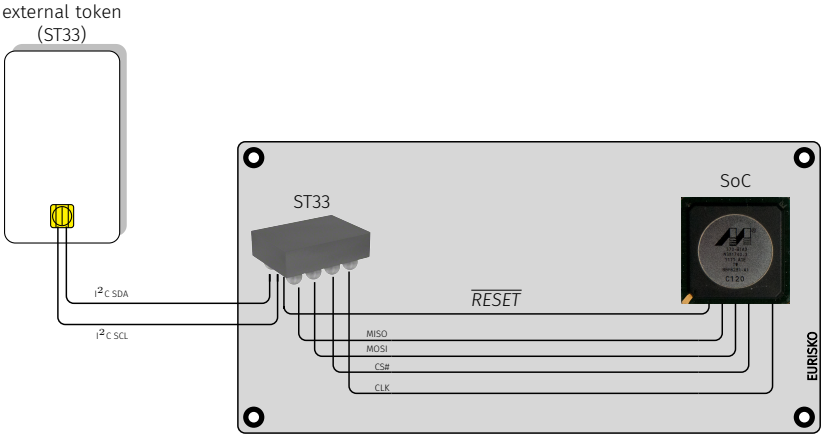
libecc *usecases*: some *examples*

Eurisko

Eurisko


external token
(ST33)
ST33
SoC
I2C SDA
I2C SCL
RESET
MISO
MOSI
CS#
CLK
EURISKO

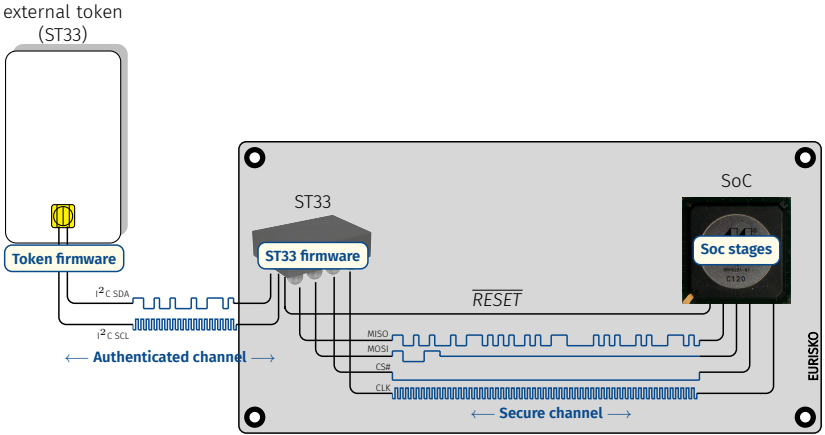
Eurisko


external token
(ST33)
ST33
SoC
I2C SDA
I2C SCL
RESET
MISO
MOSI
CS#
CLK
EURISKO

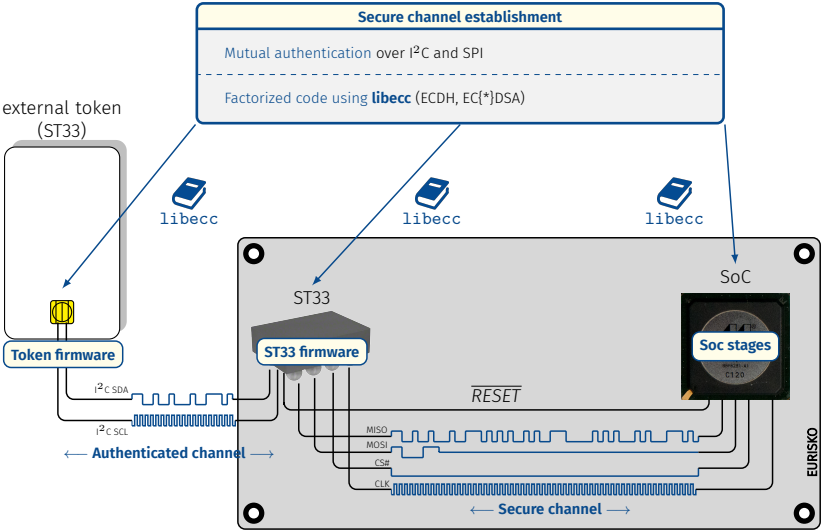
EURISKO

Secure channel establishment

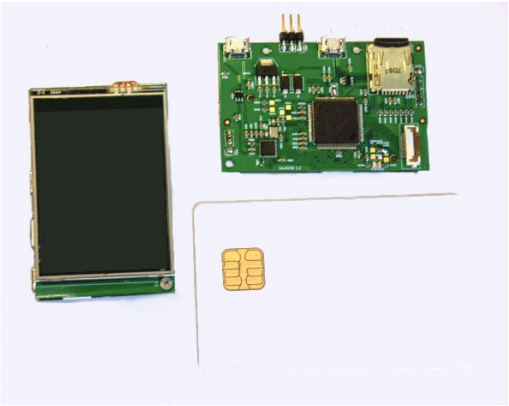
Mutual authentication over I²C and SPI



Eurisko


Secure channel establishment
Mutual authentication over I²C and SPI
Factorized code using libecc (ECDH, EC{*}DSA)
external token
(ST33)
libecc
libecc
libecc
Token firmware
ST33
ST33 firmware
SoC
Soc stages
I²C SDA
I²C SCL
RESET
MISO
MOSI
CS#
CLK
Authenticated channel
Secure channel
EURISKO

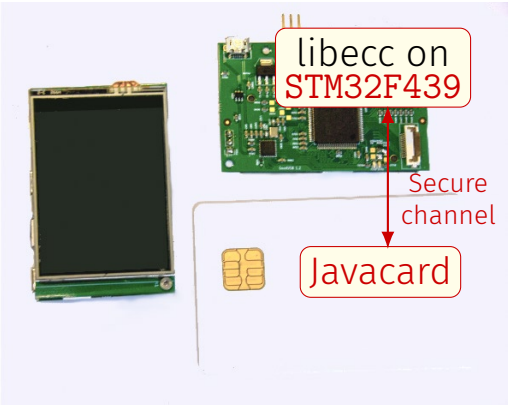
The WooKey project



open hardware

open source

The WooKey project



open hardware

open source

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!