Attacking embedded ECC implementations through cmov side channels

Erick Nascimento[1] Łukasz Chmielewski[2] David Oswald[3] Peter Schwabe[4]

[1]University of Campinas, Campinas, Brazil

[2]Riscure BV, Delft, The Netherlands

[3]University of Birmingham, Birmingham, UK

[4]Radboud University, Nijmegen, The Netherlands

Memorial University of Newfoundland, St John's, NL, Canada
Aug 12, 2016

Overview

1. **Introduction**
2. **CSWAP implementations**
3. **Attack setup**
4. **Attacks**
5. **Error correction**
6. **Countermeasures**

Outline

1. Introduction
2. CSWAP implementations
3. Acquisition Setup
4. Attacks
5. Error detection and correction
6. Countermeasures
7. End

Curve25519

- Bernstein proposed Curve25519 and the associated X25519 Diffie-Hellman Key Exchange protocol in 2006.
- Curve25519 is an elliptic curve in the Montgomery form with equation

$$E(\mathbb{F}_p) : y^2 = x^3 + 48662x^2 + x$$

  over the prime finite field $\mathbb{F}_p$, $p = 2^{255} - 19$ (pseudo-Mersenne).
- For efficiency, field elements are usually represented modulo $2p = 2^{256} - 38$, and reduced modulo p only when necessary.

X25519: Curve25519's key agreement scheme

- 128-bit security.
- We focus on the variable-base ECSM, for computing the shared secret.
- Firstly, the secret scalar is "clamped".
- Then, a variable-base scalar multiplication $R \leftarrow [k]P$ is computed, where k is a clamped secret scalar and P is a (variable) point.
- Output is the x-coordinate x_R of point R.
- Several ECSM algorithms can be applied to Curve25519, but the Montgomery Ladder is the most widely used, due to fast XZ-coordinates arithmetic due to Montgomery's differential addition formulas.

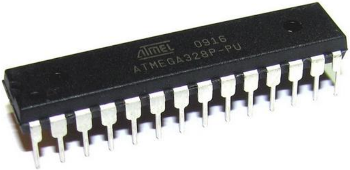
X25519: Montgomery Ladder

Algorithm 1 Montgomery ladder for Curve25519

Input: 255-bit scalar s, x-coordinate x_P of P.
Output: $(X_{[s]P}, Z_{[s]P})$, such that $x_{[s]P} = X_{[s]P}/Z_{[s]P}$.

$P1 \leftarrow (x_P, 1)$; $P2 \leftarrow PDbl(P1)$. {Because $s_{254} = 1$ after clamping}
for $i \leftarrow 253$ **downto** 0 **do**
    if $s_i = 1$ **then**
        $P1 \leftarrow PAdd(P1, P2, x_P)$
        $P2 \leftarrow PDbl(P2)$
    else
        $P2 \leftarrow PAdd(P1, P2, x_P)$
        $P1 \leftarrow PDbl(P1)$
    end if
end for
return $P1$

Target device: ATmega328P



- 8-bit RISC microcontroller.
- AVR is a Harvard-based architecture with separate address spaces for data (SRAM), program (Flash) and non-volatile data (EEPROM).
- 32KB Flash, 2KB SRAM and 1KB EEPROM.

Timing Analysis (TA) and Simple Power Analysis (SPA)

Elapsed time typically varies and depends on the specific value of the input data being processed on the particular run.

- **"Avoid secret-dependent load addresses"**.
  - Not required for AVR, as there's no memory hierarchy.
- **"Avoid secret-dependent branch conditions"**.
  - Balance the # of cycles when branch is taken or not taken (error-prone) or apply boolean operations.

Power consumption depends on the data and operation:

- Constant time is not enough: must execute **same sequence of instructions** in every run.
- Data leakage: instruction operands should be randomized (preferably) or their Hamming Weight has to be balanced.

Attacks proposed

Propose two attacks, one against a different implementation of the CSWAP operation.

- Both are profiled, template SPA attacks.
- Reduced templates are used.
- They are single-trace attacks: one trace is enough to recover the key.

They work against implementations protected with all the typical countermeasures, such as:

- Projective coordinate randomization. Also against its stronger version, *re-randomization*.
- Scalar randomization.
- Point blinding.

ECC implementations for AVR

Name	Description	SCA countermeasures
micro-ecc	8/32/64-bit C impl. of NIST curves	apparently rand. proj. coords.
nano-ecc	Derivate of micro-ecc	same as micro-ecc
μNaCl	Curve25519 for 8/16/32-bit processors	constant-time
AVR-Crypto-Lib	ECDSA with NIST P-192	none
FLECC_IN_C	8/16/32/64-bit C impl. for various curves	constant time, rand. proj. coords.
RELIC	Various curves and fields supported	constant-time
WM-ECC	Impl. for sensor networks	none
TinyECC	Impl. for sensor networks	none
MIRACL	Lib. supporting multiple curves	none
WolfSSL	Support for AVR unclear	none
Wiselib	Lib. for distributed systems	none
CRS ECC	Commercial, closed source	none

Table: Overview of ECC implementations for AVR.

Outline

1st impl.: CSWAP of field elements (CSWAP-data) I

Algorithm 2 Montgomery ladder with arithmetic cswap and randomized projective coordinates.

```
// ... initialization omitted ..
bprev ← 0
for i = 254 ... 0 do
    re_randomize_coords(work)
    b ← bit i of scalar
    s ← b ⊕ bprev
    bprev ← b
    cswap_coords(work, s)
    ladderstep(work)
end for
```

1st impl.: CSWAP of field elements (CSWAP-data) II

Algorithm 3 Constant time arithmetic/boolean CSWAP.

Input: 8-bit words x and y, cswap bit b.
Output: x and y are swapped iff $b = 1$.

$m \leftarrow -b$ {m = 0, if b = 0; else m = 0xFF}
$t \leftarrow (x \oplus y) \bullet m$
$x \leftarrow x \oplus t$
$y \leftarrow y \oplus t$

- The CSWAP function is applied to every pair of words (32 pairs) for each pair of point coordinates, (X_1,X_2) and (Z_1, Z_2).
- For a total of 64 calls per ECSM iteration. Thus, the AND with the secret mask is also performed 64 times per ECSM iteration.

2nd impl.: CSWAP the pointers (CSWAP-pointers)

Algorithm 4 Constant time implementation of secret-dependent if/else branch.

1: Let $pP1$ and $pP2$ be pointers to $P1$ and $P2$, respectively.
2: $\mathrm{CSWAP16}(1 - s_i, pP1, pP2)$ $\{s_i$ is the scalar bit, swap if $s_i = 0\}$
3: $pP1 \leftarrow \mathrm{PAdd}(P1, P2, x_P)$
4: $pP2 \leftarrow \mathrm{PDbl}(pP2)$

- The CSWAP function is applied twice, once for each 8-bit word of the pointer value (in AVR pointers are 16-bit wide).
- Therefore, this method reduces significantly the number of ANDs with the secret mask, from 64 to 2.
- On the other hand, now the secret are the addresses pointed to by pP1 and pP2.

SPA and Template SPA Countermeasures

Highly regular ECSM algorithms implemented in constant time are insufficient, due to e.g. Horizontal Collision Attacks or DPA.

Additional countermeasures have to be applied, such as:

1. Projective coordinates randomization;
2. Scalar randomization (SR);
3. Point blinding.

The target implementation is **uNaCl for AVR**, with projective coordinates *re*-randomization applied on top of it.
No assumption is made about the scalar: work against implementations protected with other countermeasures, such as SR.

Projective Coordinates Randomization [Coron99]

Input is u, the x-coordinate of input point P.

1. Generate random $\lambda \in_R \mathbb{F}_p \backslash \{0\}$.
2. Do $Z_2 \leftarrow \lambda$ and $X_2 \leftarrow u \cdot \lambda$, where u is the x-coordinate of input point P.
3. Use $P' = (X_2 : Z_2)$ in place of P.

Can also be used at each ECSM iteration (a.k.a. re-randomization).

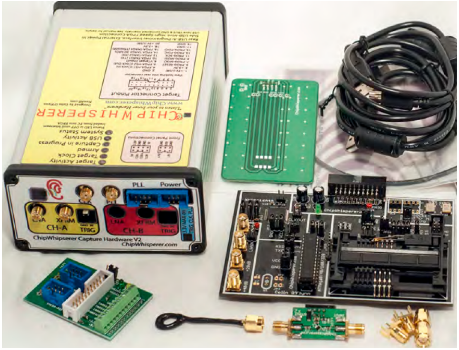
Scalar Randomization [Coron99]

At the beginning of the scalar multiplication:

1. Randomly choose $r \in \{0, 1\}^n$, for a small n. $n = 32$ seems to be a reasonable security/efficiency trade-off.
2. Compute $k' \leftarrow k + r|E|$.
3. Use k' in place of k.

Outline

ChipWhisperer and Picoscope 5203



Picoscope 5203

- Sample rate: 500 MSa/s.
- Buffer length: 32 MSa.

Acquisition

- AVR is clocked at $f_{\mathrm{dev}} = 7.3728 MHz$, 1 cycle = 135.63 ns.
- Placed a 49.9 Ohm resistor into the ground path.
- Measured using Picoscope 5203 at a sample rate $f_{\mathrm{s}} = 500 MHz$.
- Not possible to capture the full ECSM, about 2s, due to limited buffer size (32M Sa).
    - Solution: utilize the scope memory segmentation feature, one segment per ECSM iteration.

Outline

CSWAP-data implementation details

Listing 1: Conditional XOR swap.

```
    ld xx, X    ; X register points to first value
    ld yy, Z    ; Z register points to second value
    mov tt, xx
    eor tt, yy
    and tt, m   ; tt = (xx XOR yy) AND m
    eor xx, tt  ; xx = xx XOR tt
    eor yy, tt  ; yy = yy XOR tt
    st X+, xx   ; store first value
    st Z+, yy   ; store second value
```

Trace filtering, resampling, cutting and alignment

- **Filtering**: digital bandpass Butterworth filter, $f_l = 300$ kHz and $f_u = 2 \cdot f_{dev} = 14.75$MHz.
- **Resampling**: since f_{dev} is not a multiple of f_s, we first re-sampled the filtered traces to f_{rs} =493.978 MHz (1 cy = 67 samples).
- **Alignment**: pattern-based approach.
  - Selects part of the first trace as the reference, and computes the euclidean distance or correlation for each offset within a chosen range for each following trace.
  - Shifts each trace by the respective offset that minimizes the distance measure.
- **Cutting**: the filtered and aligned traces were cut into sample vectors, each corresponding to the power samples of a single instruction.
  - Based on the execution trace obtained by running the same binary in a cycle-accurate AVR simulator.
  - Enabled us to generate templates for a specific instruction or an instruction sequence with cycle accuracy.

Template-based Simple Power Analysis

1. Template **building phase** (Offline): try to characterize/profile the power consumption of a sequence of instructions executed on a device identical to the target's device.
2. Template **matching phase** (Online): matches each template against a **single** trace captured from the target device.
   - The strongest match is most likely the right one.
3. Limitations:
   - Different devices $\Rightarrow$ different power consumption characteristics.
   - Multivar. gaussian model is numerically unstable; POI selection.
   - Assumes that a known key and/or data is processed by the device, else cannot build templates.
     $\Rightarrow$ **Scalar randomization (SR)** has to be disabled in profiling phase.

TA matching, classification and estimated confidence

Classification: compute Euclidean distance between sample vector and template mean vector. The template with the smallest distance, T_0 or T_1, is considered the best match.

Confidence score (CS): derived based on distances d_0 and d_1 to each template:

$$\text{conf\_score} = 2 \cdot \left| 0.5 - \frac{min(d_0, d_1)}{d_0 + d_1} \right| \quad (1)$$

Confidence level (CL): call the recovered bit as *suspicious* if its confidence score is less than the greatest CS of any wrongly identified bit, determined in profiling phase. The CL is the percentage of bits that are not suspicious.

TA on CSWAP-data

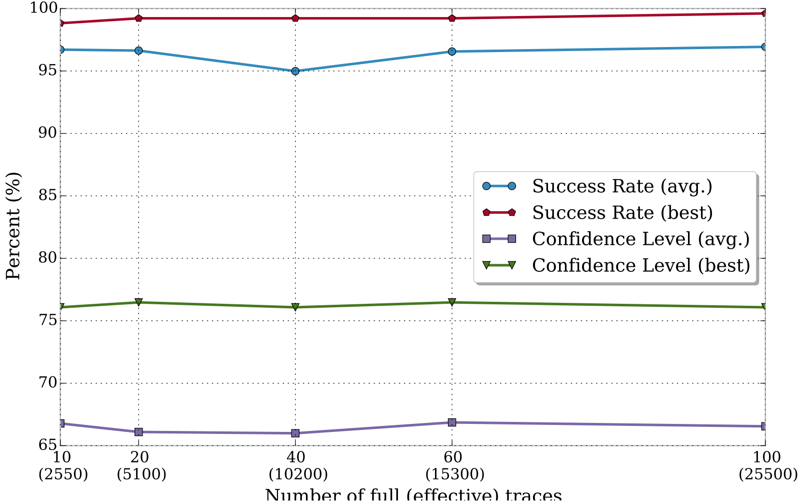


Figure: Results for the AND attack for different number of full traces (i.e., all 255

CSWAP-pointers: detail of ladderstep

Listing 2: Segment of the ladder step code.

```
static void ladderstep_explicit_coords(fe25519 *x0, fe25519
    *x1, fe25519 *z1, fe25519 *x2, fe25519 *z2)
{
/* initialization omitted */

ladderstep_1st_fp_add_x2_z2_begin:
fe25519_add(t1, xq, zq);
ladderstep_1st_fp_add_x2_z2_end:
fe25519_sub(xq, xq, zq);
ladderstep_2nd_fp_add_x1_z1_begin:
fe25519_add(zq, xp, zp);
ladderstep_2nd_fp_add_x1_z1_end:

/*...*/
}
```

CSWAP-pointers: Segment of execution trace for a field addition

Listing 3: Segment of the execution trace for a field addition.

```
0x171a: fp_add+0x5   LD R20, X+          ; first byte of a
0x171a: fp_add+0x5   CPU-waitstate
0x171c: fp_add+0x6   LD R21, Y+          ; first byte of b
0x171c: fp_add+0x6   CPU-waitstate
0x171e: fp_add+0x7   ADD R20, R21
0x1720: fp_add+0x8   ST Z+, R20          ; first byte of r
0x1720: fp_add+0x8   CPU-waitstate
```

CSWAP-pointers: Number of executed instructions of each type that are used in the attack

Table: Number of executed instructions used in the attack, grouped by type.

Type	1^{st} `fp_add`	`fp_sub`	2^{nd} `fp_add`	Total
`LD R20, X+`	32	32	16	80
`LD R21, Y+`	32	32	16	80
`LD R20, Z+0`	33	33	0	66
`ST Z+, R20`	65	65	16	146

Template attack on CSWAP-pointers (II)

Class	Method / Param. Name	Param. Value	SR (%)	CL (%)
	No filtering	-	57.3	-
	Upper cutoff freq.	2.5 * f_{dev}	92.9	-
	"	2.0 * f_{dev}	94.3	-
	"	1.7 * f_{dev}	92.9	-
	(pLow, pHigh); nPOI	(12.5, 87.5); 23	58.5	32.4
	"	(35, 65); 324	94.3	36.8
POI Selection	(pLow, pHigh); nPOI	(40, 60); 1500	64.1	31.6
	Force $\geq$ 1 sp per instr.	(35, 65); 669	92.1	68.6
	Force $\geq$ 1 sp per instr.	(40, 60); 1724	90.0	71.1
	Limit 1 sp per instr.	(35, 65); 134	85.7	8.6
	Limit 1 sp per instr.	(40, 60); 723	78.6	28.6
Classification	Sum of distances + POI	(35, 65); 324	94.3	33.9
	Majority voting + POI	(35, 65); 324	57.0	9.8
	Normal sum + POI	1; (35, 65)	94.3	38.6
	"	10; (35, 65)	92.8	36.4
Win. compression	Normal sum + POI	67; (35, 65)	79.3	20.7
	Absolute sum + POI	1; (35, 65)	94.3	23.1
	"	10; (35, 65)	92.1	27.6
	Absolute sum + POI	67; (35, 65)	77.1	18.3
	Multiple of stdev	2.0	92.1	40.7
Outlier removal	"	1.7	90.0	40.7
Distinguisher	Euclidean Distance	-	92.1	57.1
	Pearson Correlation	-	93.6	61.4
Combinations	EuclDst. + $\geq$ 1 sp per instr.	(35, 65); 669	92.1	79.3
	Corr. + $\geq$ 1 sp per instr.	(35, 65); 669	93.6	65.0

Template attack on CSWAP-pointers (III)

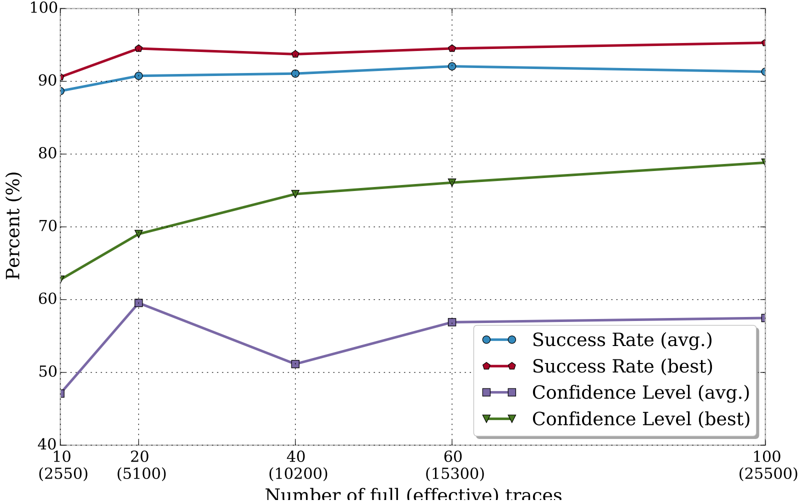


Figure: Results for the Load attack for different number of full traces (i.e., all 255

Template attack on CSWAP-pointers (IV)

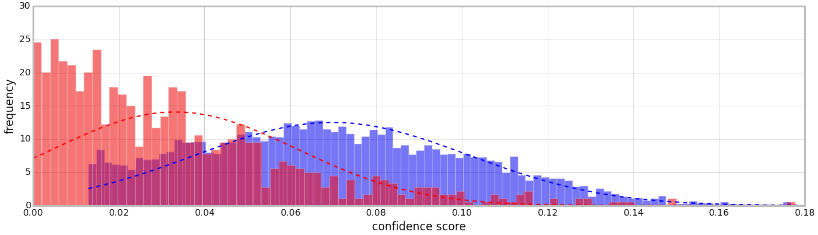


Figure: Distribution of confidence scores over all traces for suspicious bits. Red: incorrectly recovered bits, blue: correctly recovered but suspicious bits.

For the purpose of error detection, we consider bits whose confidence score is above a given threshold to be correctly recovered

Summary of Template Attacks Results

- CSWAP-data: success rate: 99.6%, confidence level: 76.1%.
  - The errors are in the cswap bits $\rightarrow$ correction is expensive per bit.
  - However, naive brute force is still feasible.
- CSWAP-pointer: sucess rate: 95.3%, confidence level: 78.8%.
  - The errors are in the scalar bits themselves $\rightarrow$ less expensive per bit.
  - Naive brute force is not feasible, as there are 54 suspicious bits to be recovered.
- Source code of targeted implementations will be available at `https://github.com/enascimento/sac2016-avr-target-impls`

Outline

Error correction / key recovery step I

Apply key recovery algorithm from [Gopalakrishnan07] for (EC)DLP-based cryptosystems with randomly located errors.

- Time-memory trade-off.

Split the partially known scalar in two parts, with arbitrary sizes:
$s = a + b \cdot 2^n$, where n is the bitlength of a.
Points P and R are an input and output pair assumed to be known to the attacker.

$$R - [b]P = [a]P \tag{2}$$

Build a table with all possibilities for the susp. bits in the lower part (a).
Try all possible values for the susp. bits in the upper half (b) (search phase), for each try a query is made to the point table.

Error correction / key recovery step II

- Time complexity is reduced from 2^{54} to $2 \cdot 2^{27}$, if scalar is splitted in half.
- Constants matter, implementation has to be efficient to achieve feasible times. For that, techniques for efficient curve and field arithmetic are employed.
- We implemented it as a single thread program. According to our estimates, 18 days are required to correct 60 errors of a 255-bit scalar.
- Source code of key recovery will be available at `https://github.com/enascimento/SCA-ECC-keyrecovery`.

Outline

Countermeasures

2015	Negre and Perin	Split the scalar in two, interleave two ECSMs	Not effective
2002	Itoh et al	Store sensitive vars in addresses with the same HW	Might mitigate
2003	Itoh et al	Randomize the memory accesses	Possibly not effective
2009	Izumi et al	Idem	Idem
2010	Izumi et al	Idem	Idem
-	ours	Allocate sensitive vars in rand. addresses	Might mitigate
2012	Heyszl et al	Swap vars at the end of each iteration	Might mitigate
2015	Le et al	Seq. of operations are indep. from scalar	Might mitigate

Outline

Thanks

Thank you for your attention!

Questions

Questions?