DE LA RECHERCHE À L'INDUSTRIE

cea

www.cea.fr

leti & list

Runtime Code Generation for Performance and Security in Embedded Systems

Damien Couroussé

2015-10-16

Univ. Grenoble Alpes, F-38000 Grenoble, France
CEA-LIST, MINATEC Campus, F-38054 Grenoble, France

Henri-Pierre Charles, Thierno Barry, Fernando Endo, Bruno Robisson, Hassan Noura, Thibault Cattelani, Caroline Queva, Charles Aracil…

COGITO people : Jean-Louis Lanet, Bruno Robisson, Thierno Barry, Philippe Jaillon, Olivier Potin, Hélène Le Bouder…

Runtime Code Generation: Motivation

Pitch: some code optimisations are not accessible to static compilers

- Unknown data
- Sometimes, the hardware is also unknown, at least partially

- Delay code optimisations at runtime
  - Constant propagation, elimination of dead code,
  - Strength reduction,
  - Loop unrolling,
  - *Instruction scheduling,*
  - etc.

(runtime) code specialisation

- Drive code performance by runtime-changing constraints
  - Bounds : power / energy / execution time
  - Heterogeneous cores : accelerators, specialised instructions

- Runtime code generation for unusual purposes (e.g. security)?

(runtime) code polymorphism

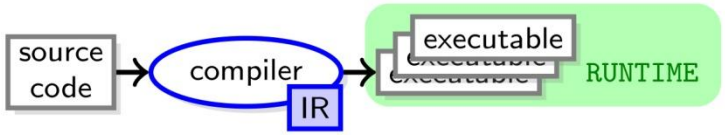
Outline

Tools for runtime code generation in embedded systems

- … for performance
- … for security

Approaches for code specialisation

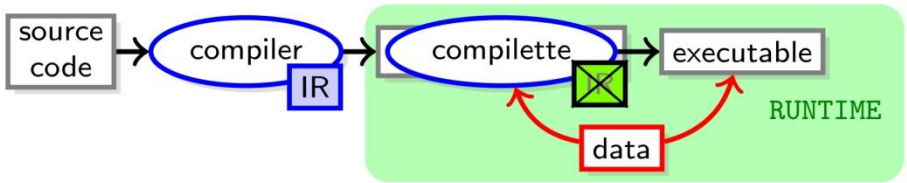
Static code versionning (e.g. C++ Templates)



- static compilation
- runtime: select executable
- memory footprint ++
- limited genericity
- runtime blindness

Runtime code generation, with `deGoal`

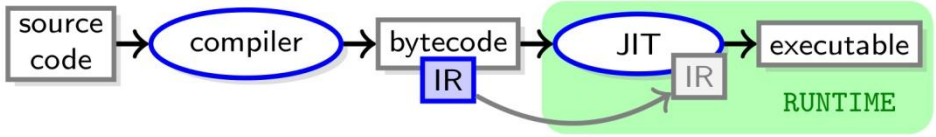
A *compilette* is an ad hoc code generator, targeting one executable



- fast code generation
- memory footprint −−
- **data-driven code generation**

Dynamic compilation

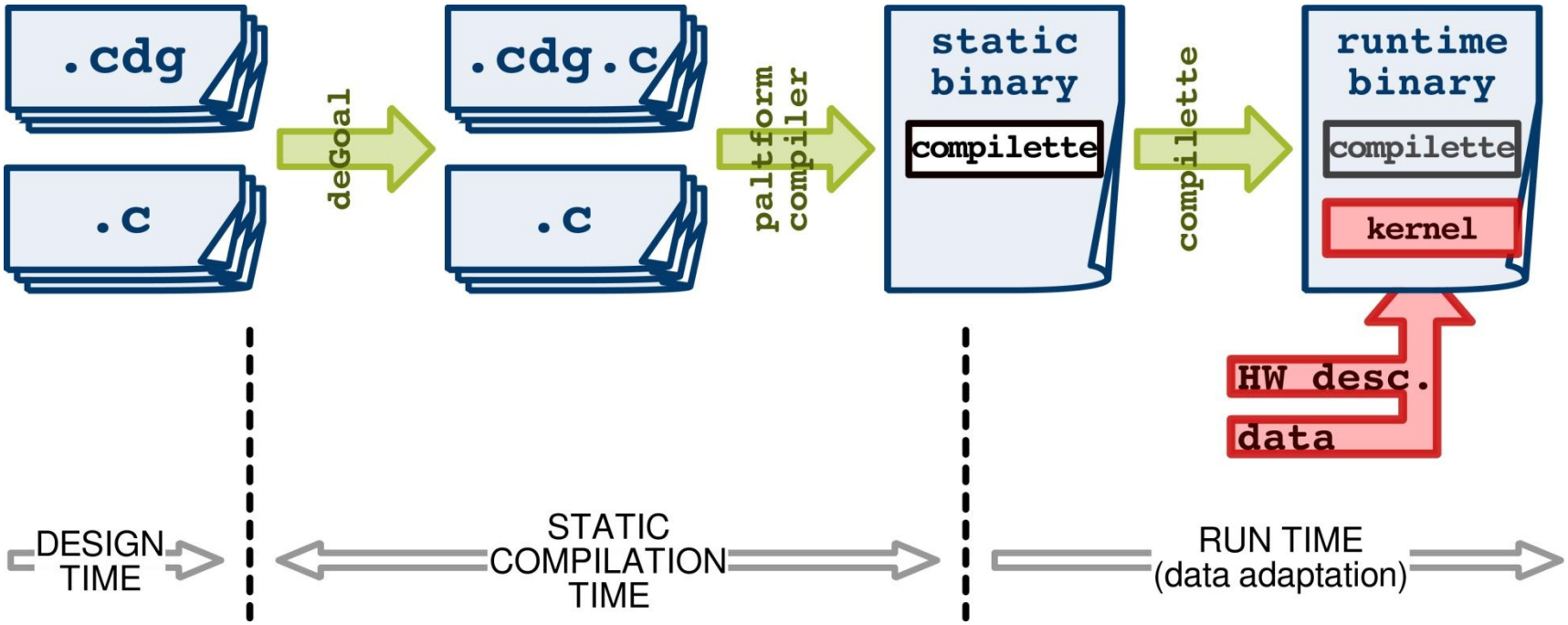
(JITs, e.g. Java Hotspot)



- overhead ++
- memory footprint ++
- not designed for data dependant code-optimisations

IR Intermediate Representation

Code generation flow



Example of deGoal code

- Simple program example: vector addition

```
void gen_vector_add(void *buffer, int vec_len, int val)
{
#[
    Begin buffer Prelude vec_addr

    Type int_t int 32 #(vec_len)
    Alloc int_t v

    lw v, vec_addr
    add v, v, #(val)
    sw vec_addr, v
]#
}
```

deGoal DSL:
Source to source converted to standard C code

Standard C code

Example of deGoal code

- Simple program example: vector addition

```
void gen_vector_add(void *buffer, int vec_len, int val)
{
#[
    Begin buffer Prelude vec_addr

    Type int_t int 32 #(vec_len)
    Alloc int_t v

    lw v, vec_addr
    add v, v, #(val)
    sw vec_addr, v
]#
}
```

When executed

Program memory:

```
ldr r1, [r0]
add r1, #1
str r1, [r0]
add r0, #4
ldr r2, [r0]
add r2, #1
str r2, [r0]
add r0, #4
```

Example of deGoal code

- Simple program example: vector addition

```
void gen_vector_add(void *buffer, int vec_len, int val)
{
#[
    Begin buffer Prelude vec_addr

    Type int_t int 32 #(vec_len)
    Alloc int_t v

    lw v, vec_addr
    add v, v, #(val)
    sw vec_addr, v
]#
}
```

Interface: pointer to code buffer and I/O registers

Type definitions and variable allocations

Instructions

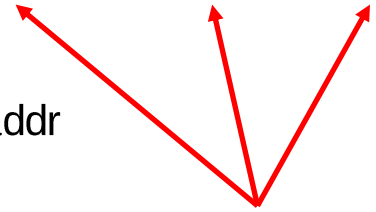
Example of deGoal code

- Simple program example: vector addition

```
void gen_vector_add(void *buffer, int vec_len, int val)
{
#[
    Begin buffer Prelude vec_addr

    Type int_t int 32 #(vec_len)
    Alloc int_t v

    lw v, vec_addr
    add v, v, #(val)
    sw vec_addr, v
]#
}
```



Determined by the application
and fixed in the final machine code

Example of deGoal code

- Simple program example:

```
void gen_vector_add(void *buffer, int vec_len, int val)
{
#[
    Begin buffer Prelude vec_addr

    Type int_t int 32 #(vec_len)
    Alloc int_t v

    lw v, vec_addr
    add v, v, #(val)
    sw vec_addr, v
]#
}
```

Example of deGoal code

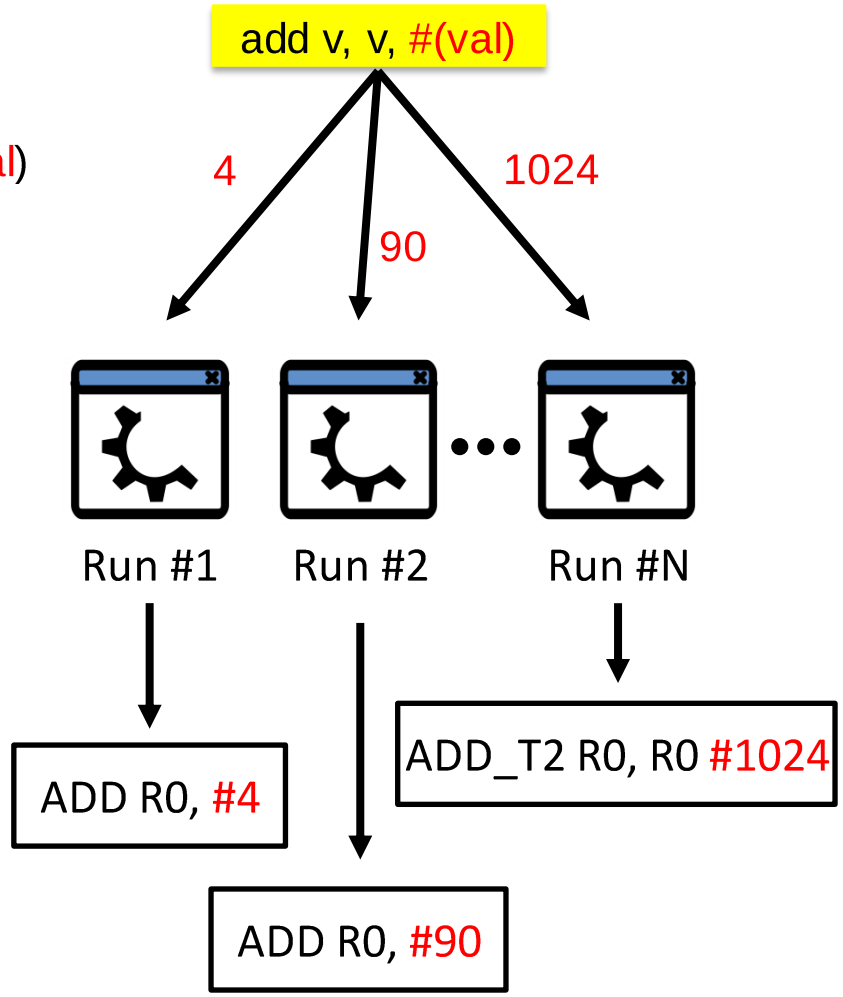
- Simple program example: vector addition

```
void gen_vector_add(void *buffer, int vec_len, int val)
{
#[
    Begin buffer Prelude vec_addr

    Type int_t int 32 #(vec_len)
    Alloc int_t v

    lw v, vec_addr
    add v, v, #(val)
    sw vec_addr, v
]#
}
```

Inline run-time constants



Example of deGoal code

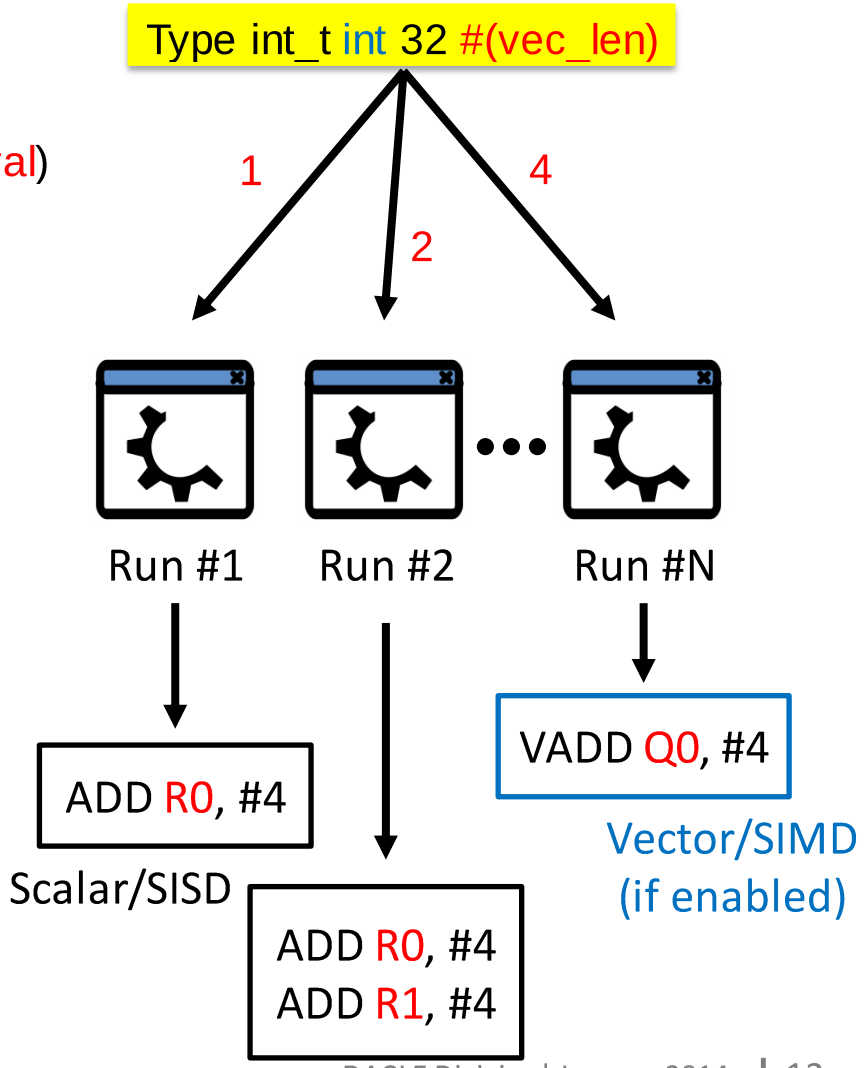
- Simple program example: vector addition

```
void gen_vector_add(void *buffer, int vec_len, int val)
{
#[
    Begin buffer Prelude vec_addr

    Type int_t int 32 #(vec_len)
    Alloc int_t v

    lw v, vec_addr
    add v, v, #(val)
    sw vec_addr, v
]#
}
```

Inline run-time constants



Example of deGoal code

- Simple program example: vector addition

```
void gen_vector_add(void *buffer, int vec_len, int val)
{
#[
    Begin buffer Prelude vec_addr

    Type int_t int 32 1
    Alloc int_t r
]#
    int i;
    for (i = 0; i < vec_len; ++i) {
#[
        lw r, vec_addr
        add r, r, #(val)
        sw vec_addr, r
        add vec_addr, #(sizeof(int))
]#
    }
}
```

Loop unrolling:
"Copy-paste" a block of instructions

Features

deGoal

- Portable DSL
- Complex variables (registers)
  - Typed
  - Vector support, dynamically sized
- Mix runtime data & binary code
- Easily extended with domain-specific or hardware-specific instructions (e.g. multimedia)

Results

- Auto-adaptative dynamic libraries
- Runtime portable optimization
- Multiple performance metrics:
  - Faster generated code
  - Smaller generated code
  - Code generation 3 order of magnitude faster than JIT/LLVM
  - Code generators 4 orders of magnitude smaller than JITs/LLVM

deGoal supported architectures

ARCHITECTURE	STATUS	FEATURES
ARM32	✓	
ARM Cortex-A, Cortex-M [Thumb-2, VFP, NEON]	✓	SIMD, [IO/OoO]
STxP70 [including FPx] (STHORM / P2012)	✓	SIMD, VLIW (2-way)
K1 (Kalray MPPA)	✓	SIMD, VLIW (5-way)
PTX (Nvidia GPUs)	✓	
MIPS	↺	32-bits
MSP430 (TI microcontroler)	✓	Up to < 1kB RAM
CROSS CODE GENERATION supported (e.g. generate code for STxP70 from an ARM Cortex-A)		

[IO/OoO]: Instruction scheduling for in-order and out-of-order cores

Outline

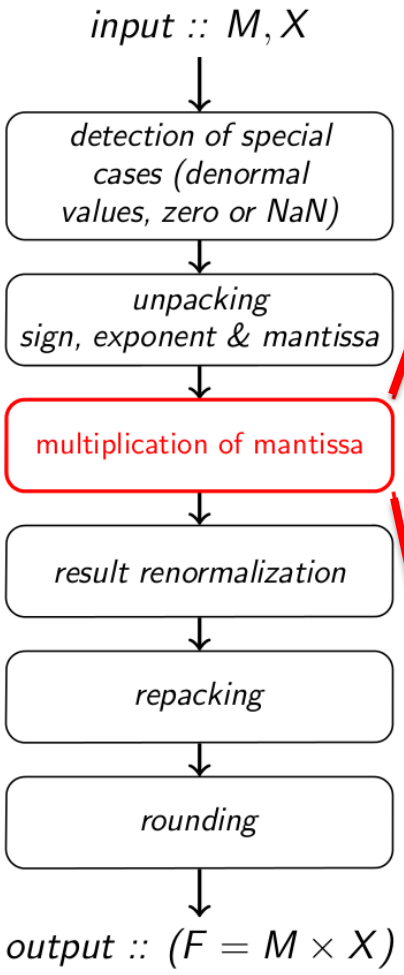
Tools for runtime code generation in embedded systems

- … for performance
  - **Application use cases**
- … for security

- Video compression (Itanium)
- Linear algebra on GPUs (Nvidia)
- Arithmetic computing on Micro-controllers for the IoT (ARM Cortex-M, TI MSP430)
- Auto-tuning for embedded applications (ARM Cortex-M + VFP + NEON)
- Memory allocation in MPSoCs
- …

Arithmetic computation on micro-controller

floating-point multiplication



F = mul(M, X)

ALGORITHM 1: Floating-Point Multiplication with Horner scheme

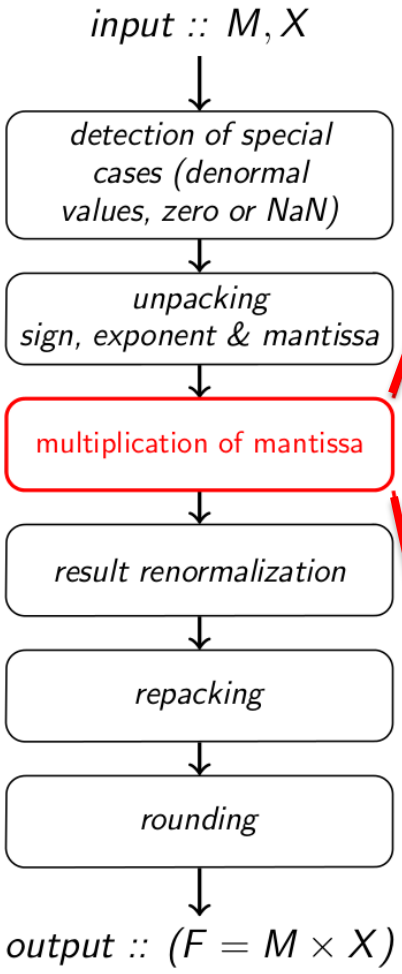
Input: Floating-point operands M and X to be multiplied (M is known, X is unknown).
Output: The result $M \times X$ of the multiplication.

```
i ← 0;
detection ← 1;
result ← X;
while not (mantissa(M) && (1 ≪ i)) do
|   i ← i + 1;
end
for i ← i + 1 to len(mantissa(M)) do
|   if mantissa(M) && (1 ≪ i) then
|   |   result ← (result ≪ detection) + X;
|   |   detection ← 1;
|   end
|   else
|   |   detection ← detection + 1;
|   end
end
result ← (result ≪ detection);
return result
```

Arithmetic computation on micro-controller

floating-point multiplication

mulM = gen(M) ➔ F = mulM(X)



ALGORITHM 1: Floating-Point Multiplication with Horner scheme

Input: Floating-point operands M and X to be multiplied (M is known, X is unknown).

Output: The result $M \times X$ of the multiplication.

$i \leftarrow 0$;

$detection \leftarrow 1$; **Kst**

$result \leftarrow X$;

while *not* $(mantissa(M)$ && $(1 \ll i))$ **do**

  $i \leftarrow i + 1$;

end

for $i \leftarrow i + 1$ **to** $len(mantissa(M))$ **do**

  if $mantissa(M)$ && $(1 \ll i)$ **then**

    $result \leftarrow (result \ll detection) + X$;

    $detection \leftarrow 1$;

  end

  else

    $detection \leftarrow detection + 1$;

  end

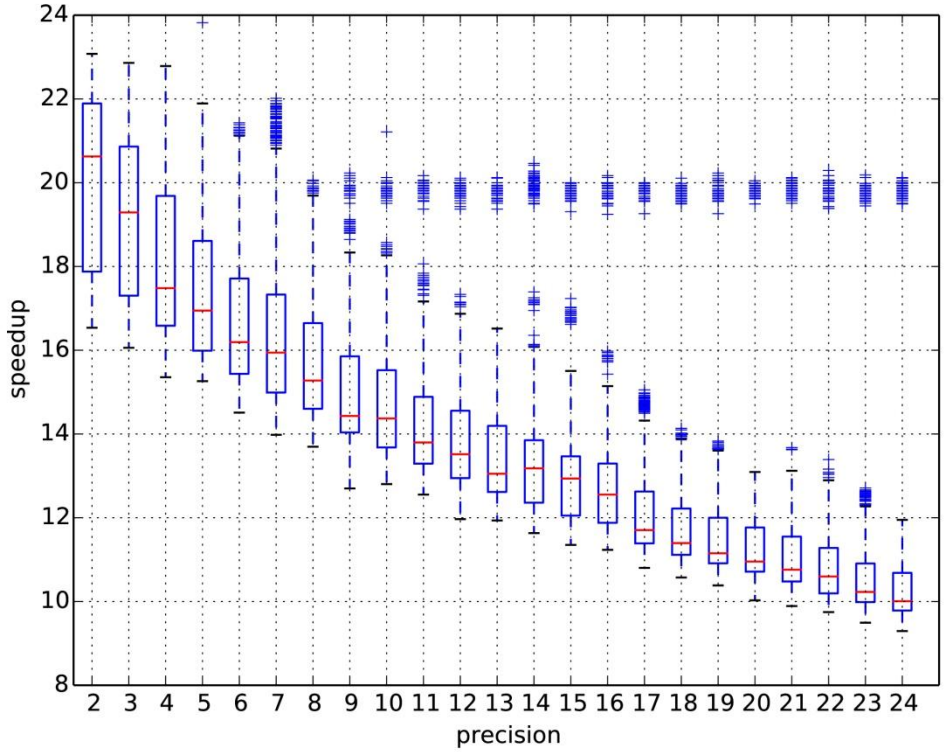
end

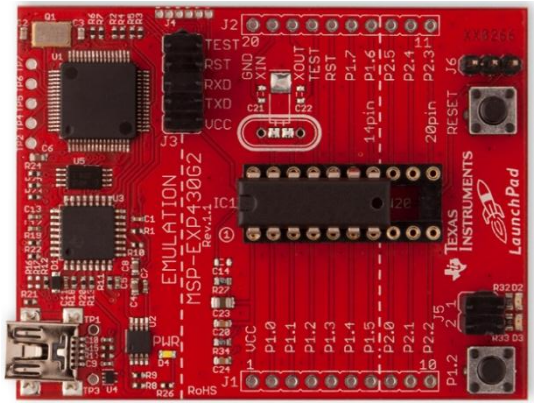
$result \leftarrow (result \ll detection)$;

return *result*

Arithmetic computation on micro-controller

- Target platform: TI MSP430
  - → **512 *bytes* of RAM!! :)**
- Specialisation
  - one operand of the multiplication
  - bit precision (width of the mantissa)
- Speedup over 10x
- Overhead recovery < 4
- Genericity: variable precision





Runtime code specialisation of floating-point arithmetics

Outline

Tools for runtime code generation in embedded systems

- … for performance
- … for security

Runtime code generation as a Software Protection for Embedded Systems against Physical Attacks

Physical Attacks

An attack is usually split between:

1. a first step attack:
   - global inspection of the target
   - identification of the security components involved (HW/SW)
   - identification of weaknesses
2. a second step attack:
   - focused attack
   - on an identified potential weakness

A coarse typology of physical attacks

- **Reverse engineering**
  - HW inspection: decapsulation, abrasion, chemical etching, memory extraction, etc.
  - SW inspection: debug, memory dumps, code analysis, etc.

- **Side channel attacks**: SPA (Simple Power Analysis), DPA (Differential –), CPA (Correlation –). . .
  - Electromagnetic analysis
  - Power analysis
  - Acoustic analysis
  - Timing attacks

  → **spatial and temporal sensibility**

- **Fault injection attacks**

  → **spatial and temporal sensibility**
  - under/over voltage drops
  - ion / laser beam, optical illumination
  - glitch attacks
  - …

Code polymorphism

Definition

- Regularly **changing the behavior** of a (secured) component, **at runtime**, while maintaining **unchanged** its **functional properties**, with runtime code generation

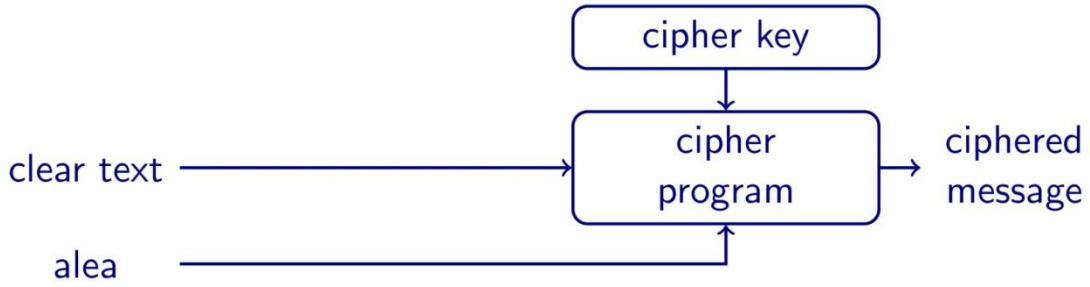
What for?

- Protection against reverse engineering of SW
  - the secured code is not available before runtime
  - the secured code regularly changes its form (code generation interval ω)
- Protection against physical attacks
  - polymorphism changes the spatial and temporal properties of the secured code: side channel & fault attacks
  - combine with usual SW protections against focused attacks
- **Compatible with State-of-the-Art HW & SW Countermeasures**

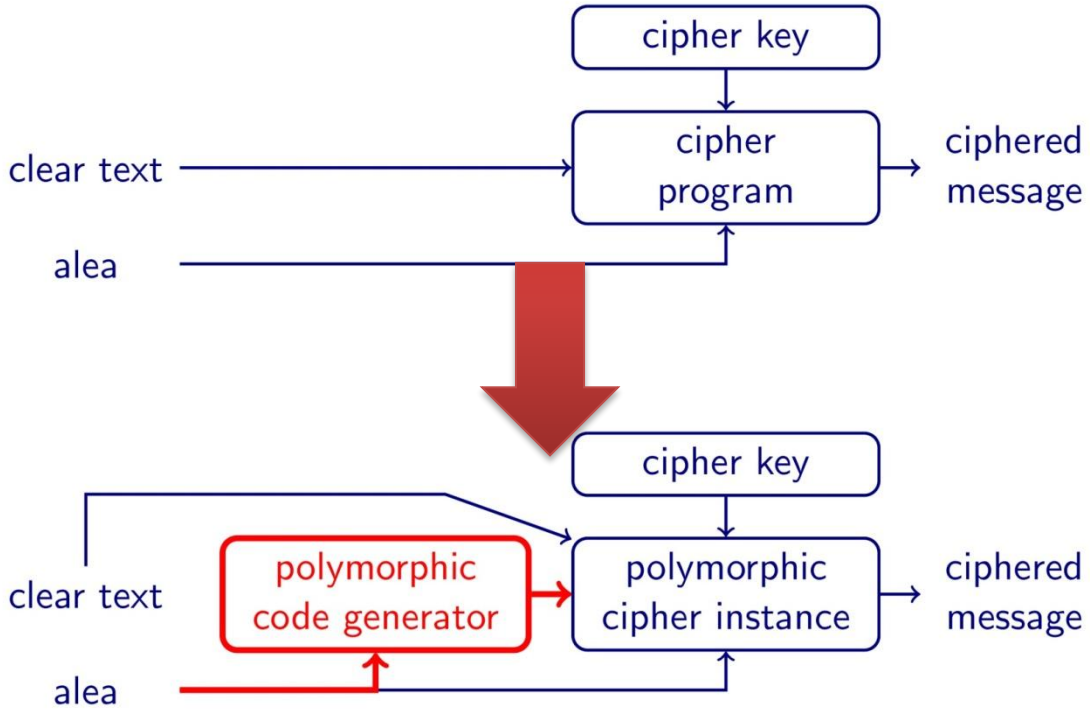
How?

- deGoal: runtime code generation for embedded systems
  - fast code generation
  - tiny memory footprint: proof of concept on TI's MSP430 (512 B RAM)

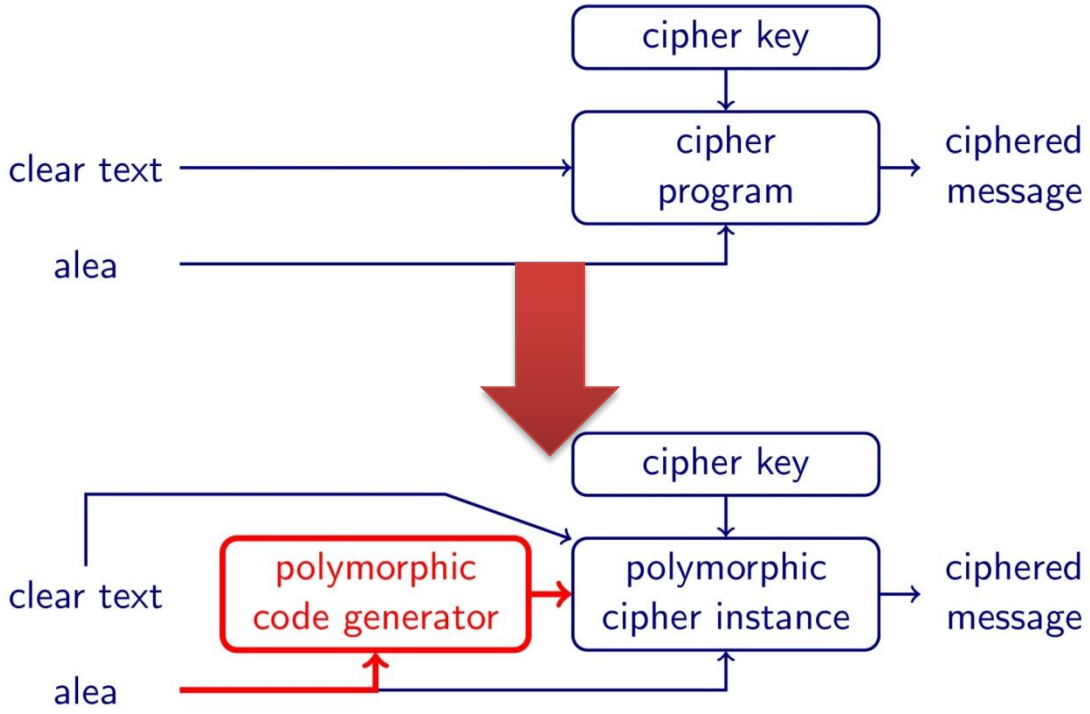
Polymorphic code generation: overview



Polymorphic code generation: overview



Polymorphic code generation: overview



Opportunities for polymorphic code generation

- Random register allocation
- Random instruction selection
- Instruction shuffling
- Insertion of noise instructions

Application to AES: AddRoundKey()

```
void addRoundKey_compilette( cdg_insn_t* code
                           , uint8_t* key_addr, uint8_t *state_addr)
{
  #[
    Begin code Prelude

    Type reg32 int 32
    Alloc reg32 state, key, i

    mv i, #(16)
    loop:
        sub i, i, #(1)
        lb state, @(#(state_addr) + i) // state = state_addr[i]
        lb key, @(#(key_addr) + i)     // key= key_addr[i]
        xor state, key
        sb @(#(state_addr) + i), state
        bneq loop, i, #(0)
    rtn
    End
  ]#;
}
```

Random register allocation

- Greedy algorithm: each register is allocated among one of the free registers remaining
- Has an impact on:
  - The management of the context (ABI)
  - Instruction selection

Demo

Instruction selection

- Replace an instruction by a semantically equivalent sequence of one or several instructions
- Select the sequence in a list of equivalences
- Examples:

```
c := a xor b <=> c := ((a xor r) xor b) xor r
c := a xor b <=> c := (a or b) xor (a and b)
c := a - b   <=> k := 1 ; c:= (a + k) + (not b)
c := a - b   <=> c := ((a + r) - b) - r
```

Instruction shuffling

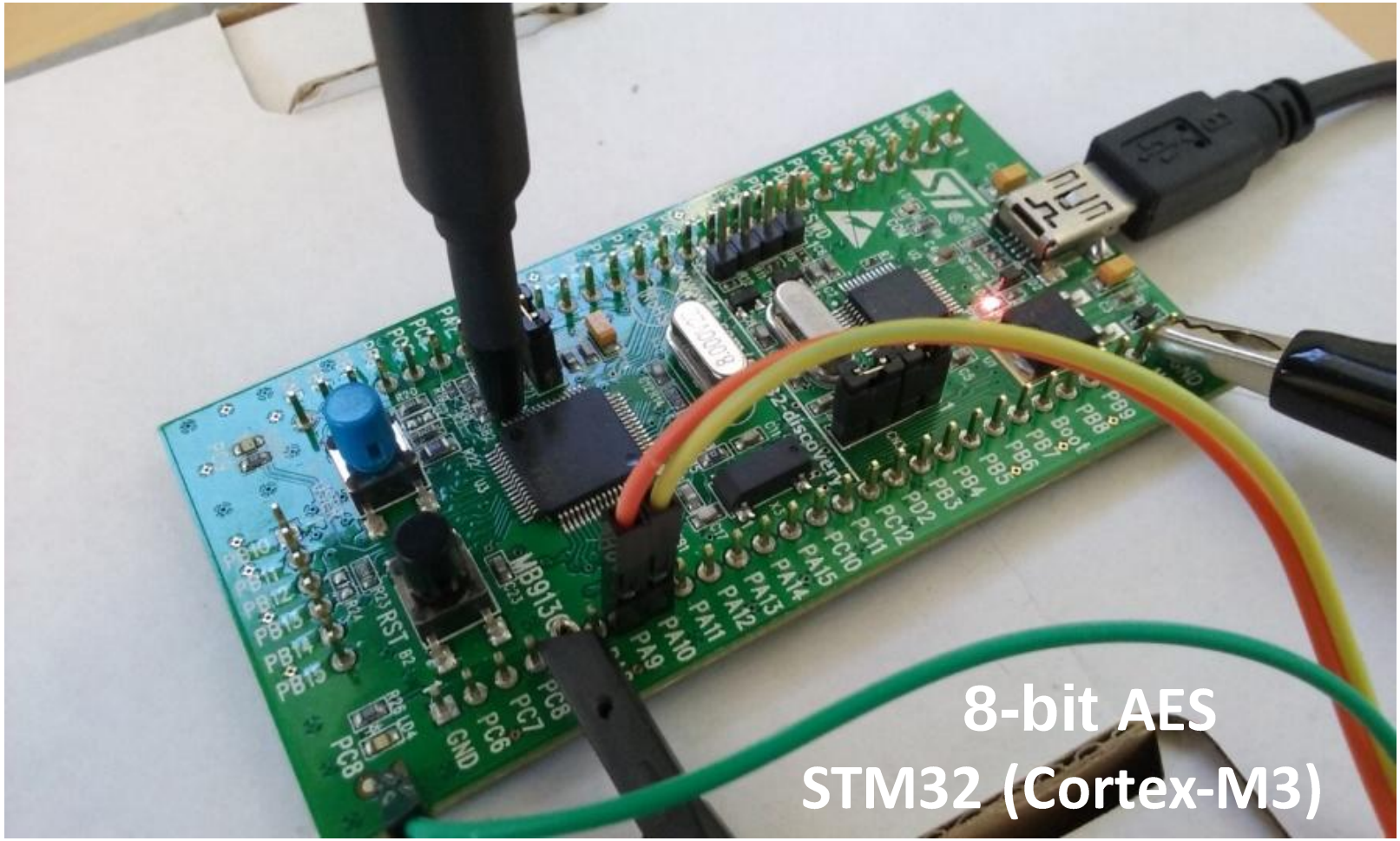
- Reorder instructions, but do not break the semantics of the code!
  - Defs
  - Uses
  - *Do not* take into account result latency and issue latency
  - Special treatments for… special instructions. E.g. branch instructions

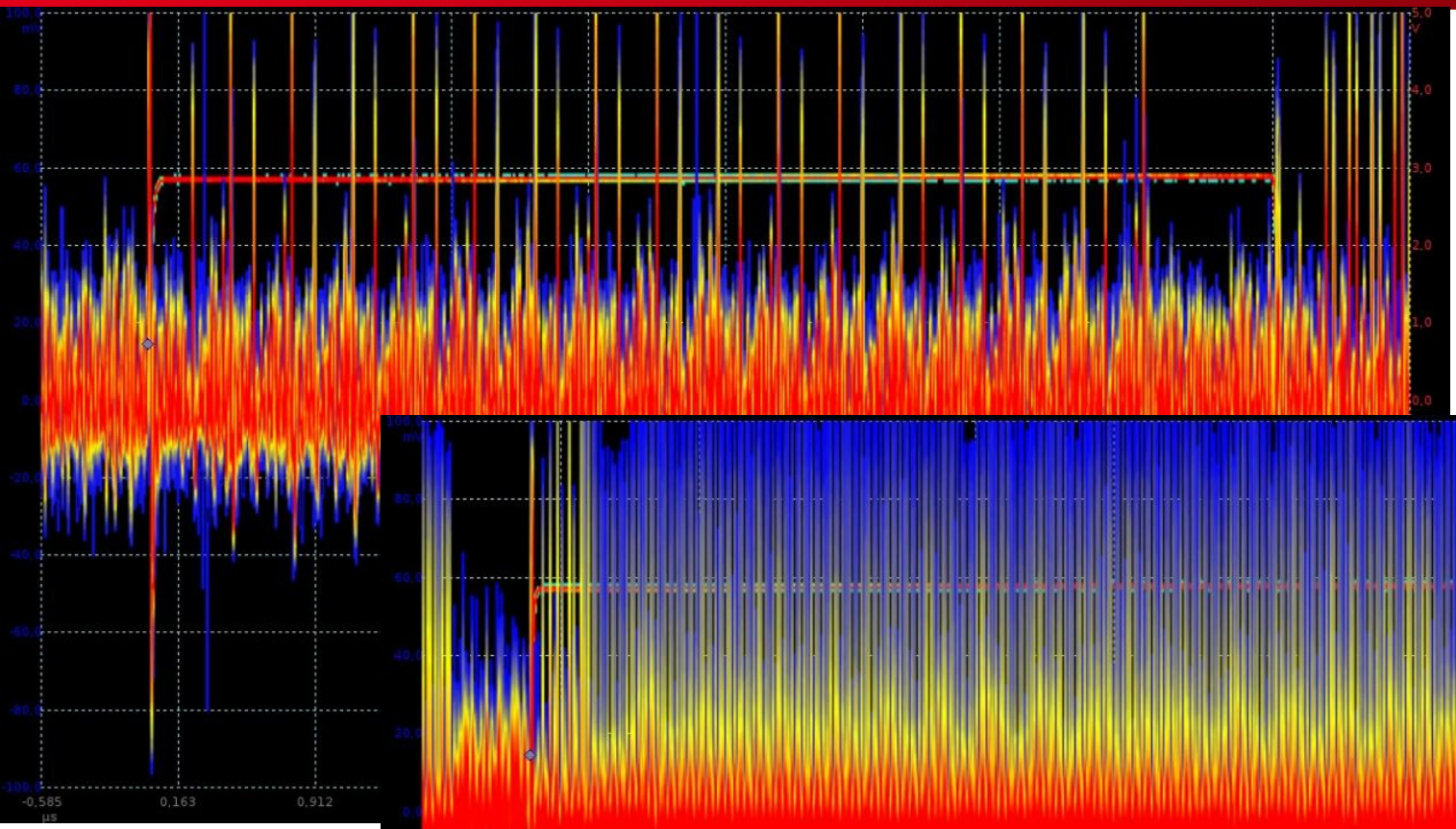
Demo

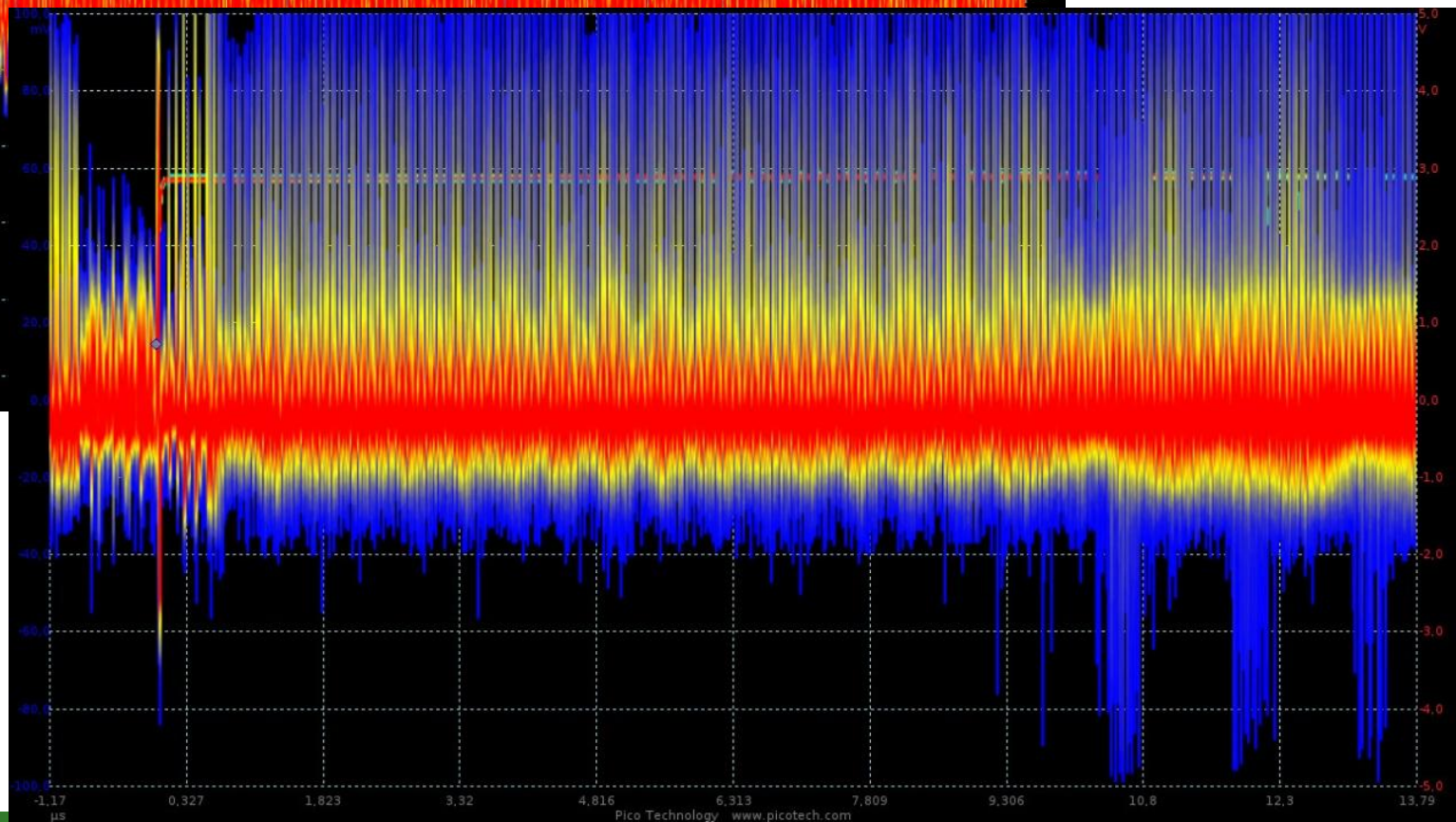
Insertion of noise

- Insertion of fake instructions, that have no effect on the result of the program.
- Controlled by a probability of insertion *p*
- Can insert any instruction:
  - Arithmetic (add, xor…)
  - Memory accesses (lw, lb, …)
  - Power hungry ones (mul, mac…)
  - nop

Illustration on AES

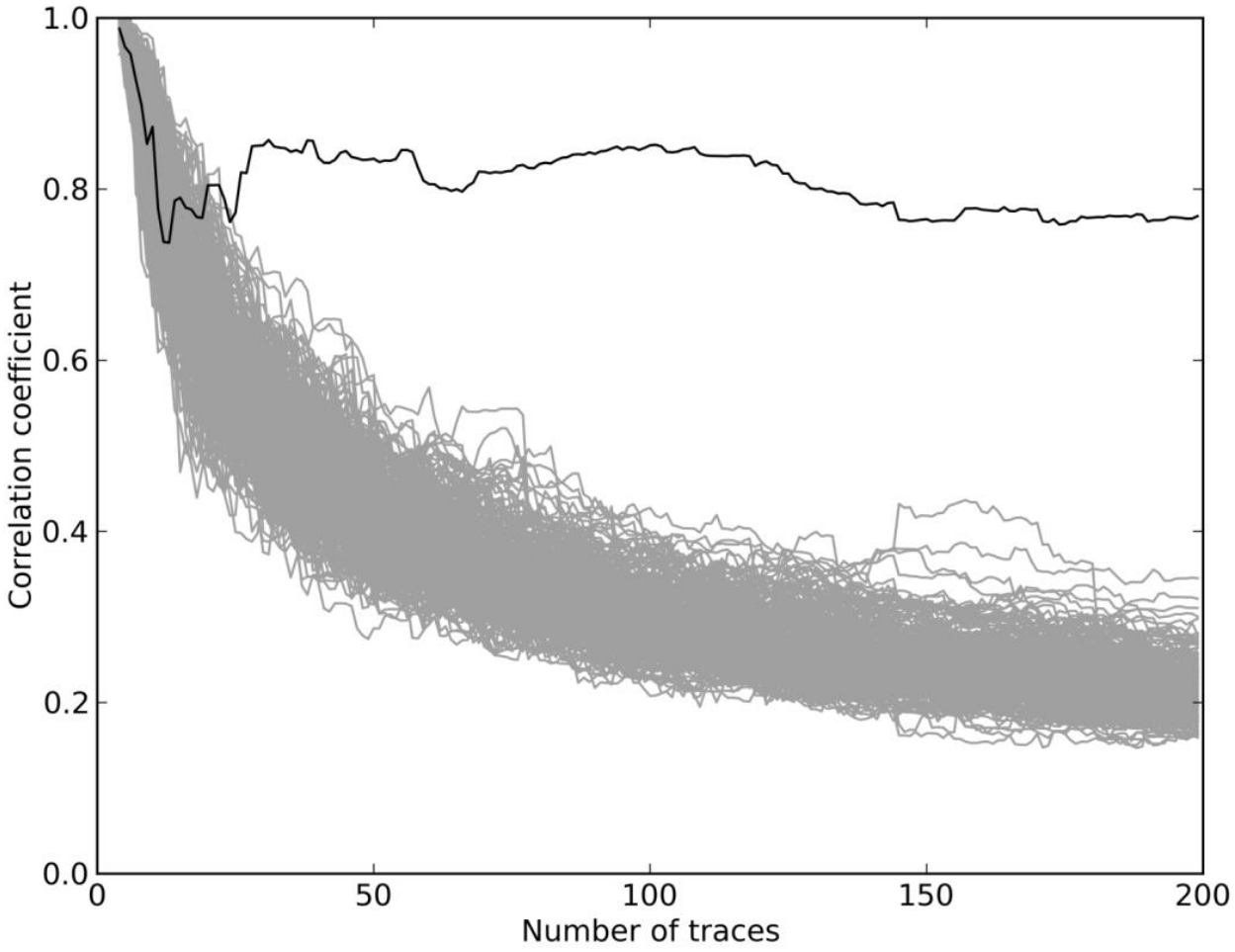



5,0
V
4,0
3,0
2,0
1,0
0,0
-0,585
µs
0,163
0,912


5,0
V
4,0
3,0
2,0
1,0
0,0
-1,0
-2,0
-3,0
-4,0
-5,0
-1,17
µs
0,327
1,823
3,32
4,816
6,313
7,809
9,306
10,8
12,3
13,79
Pico Technology www.picotech.com

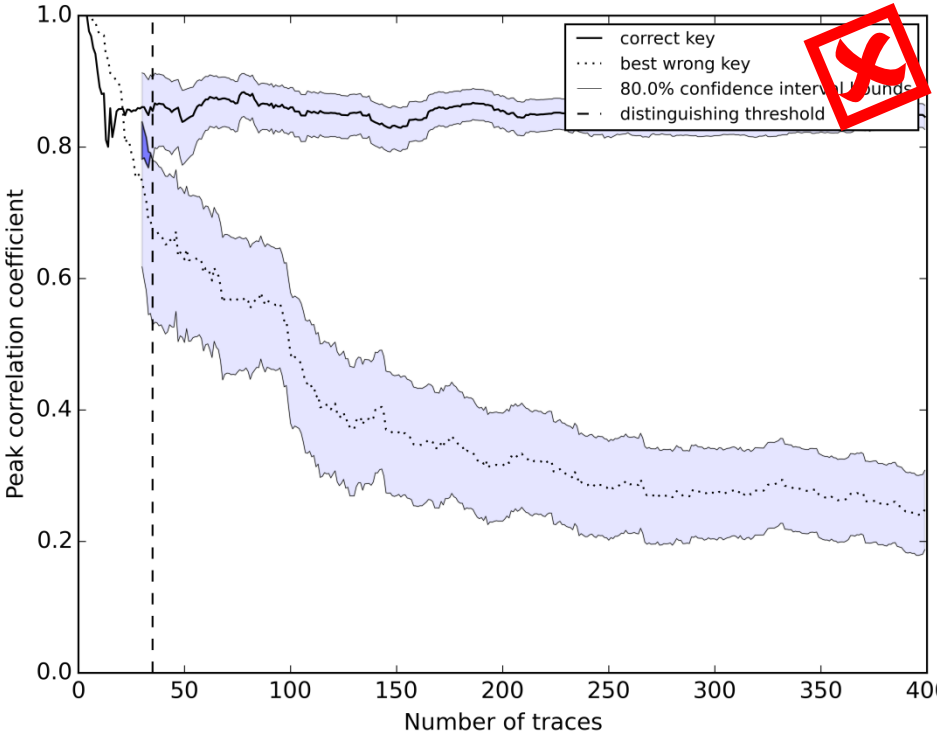
Experimental validation: CPA on EM traces

Reference implementation



Experimental validation: CPA on EM traces

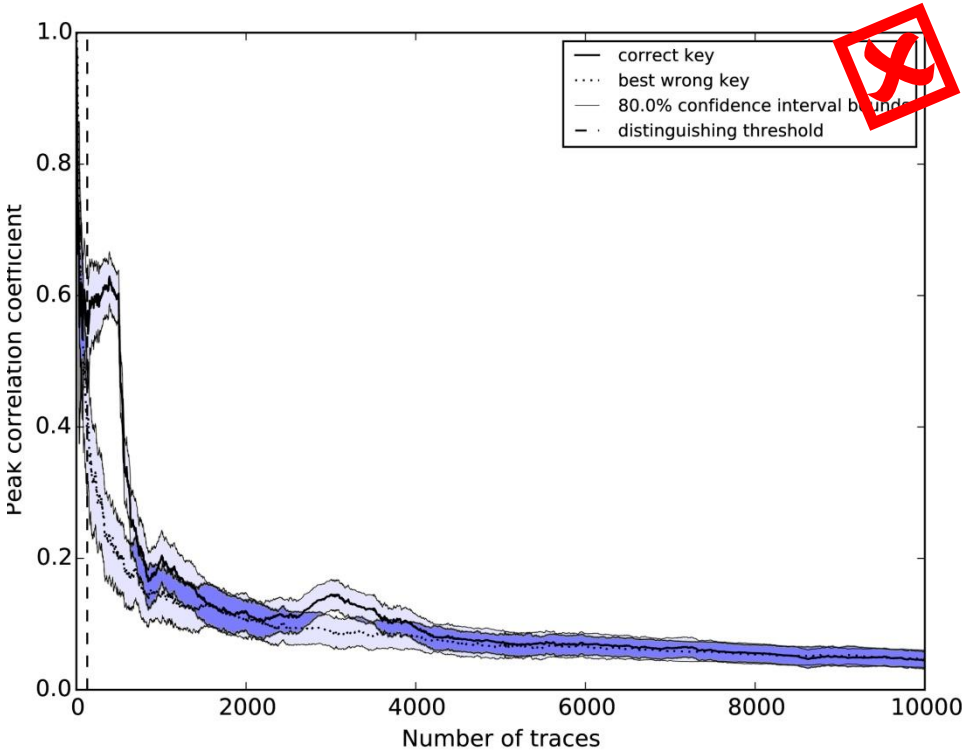
Reference implementation



Distinguish threshold = 39 traces

Key byte 10

Polymorphic version, code generation intervall: 500

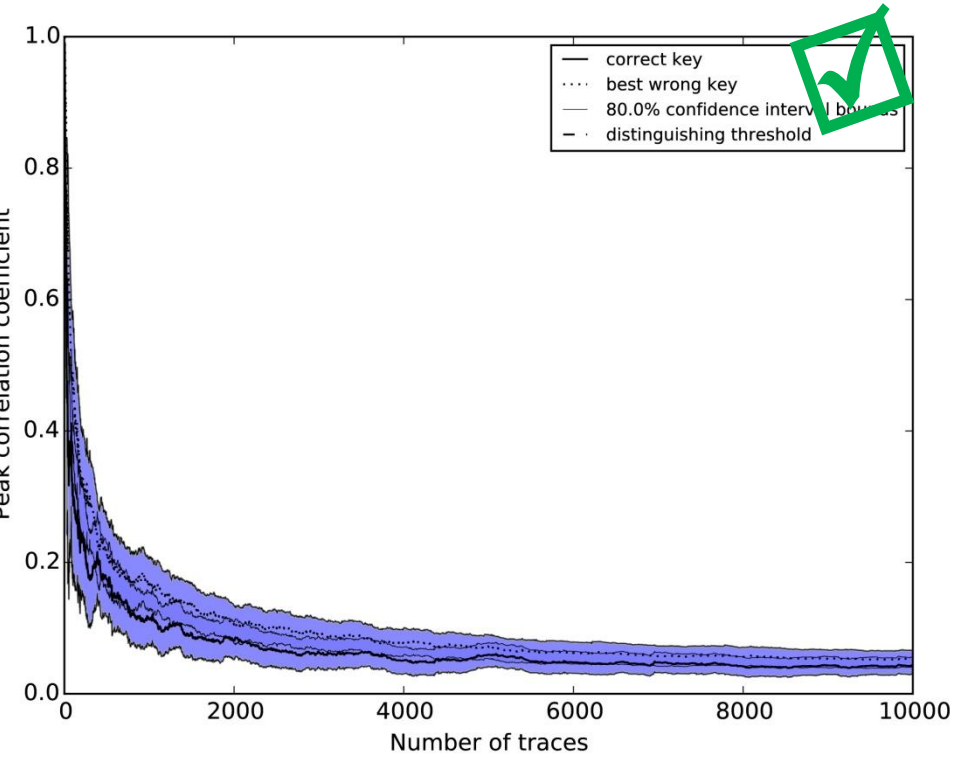


Distinguish threshold = 89 traces

Key byte 02

Experimental validation: CPA on EM traces

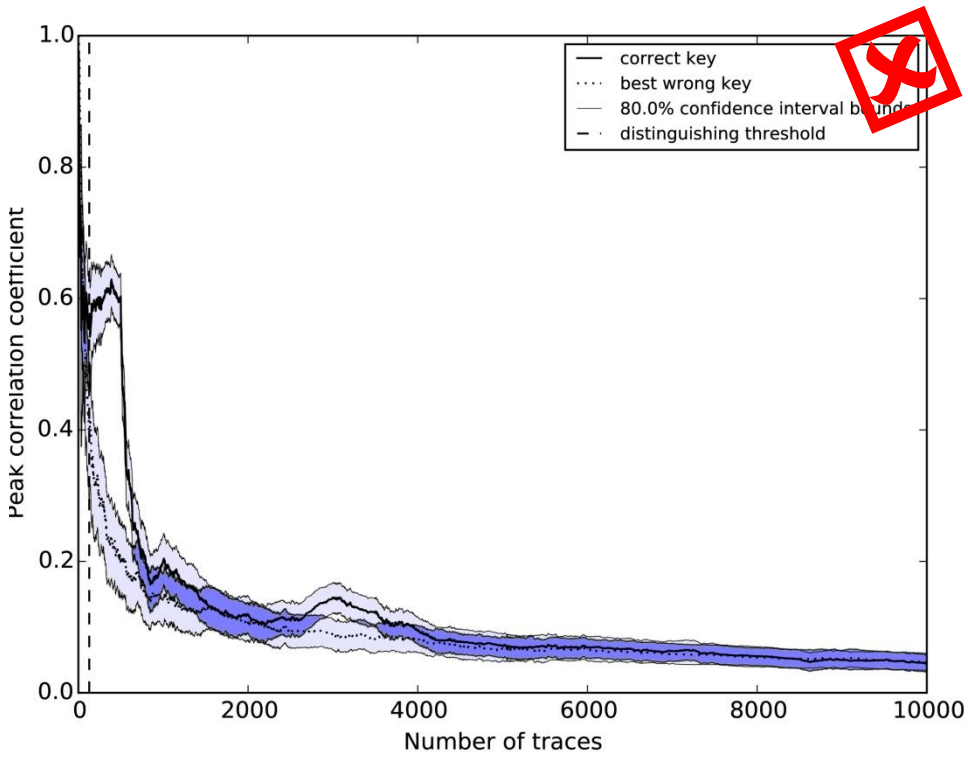
Polymorphic version
code generation interval: 20



Distinguish threshold > 10000 traces

Key byte 02

Polymorphic version,
code generation intervall: 500



Distinguish threshold = 89 traces

Key byte 02

Performance overhead

- 8 bit AES
- Polymorphic code generation:
  - AddRoundKey()
  - SubBytes()
- All overheads included

Execution times (in cycles), over 1000 runs:

	min	max	average
reference	6385	6385	6385
code generator	5671	12910	9345
polymorphic instance	7185	9745	8303

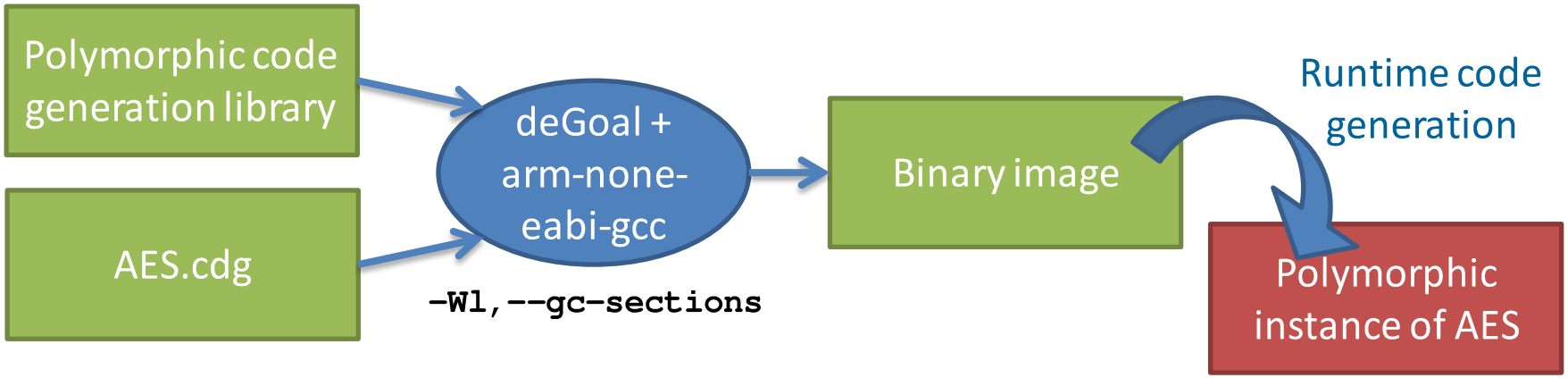
Impact of the code generation interval ω:

ω	k	%
1	2.76	53.0%
5	1.59	18.4%
20	1.37	2.1%
100	1.31	1.1%

k: overhead vs. reference implementation

%: percentage contribution of runtime code generation to the performance overhead

Memory footprint



- Polymorphic code generation library (64 variants)
  - Average: 22395,5 bytes
  - Min: 19632 bytes
  - Max: 25208 bytes
- Binary Image: size increase vs the reference implementation
  - Reference: 73279 bytes
  - Polymorphic version, max size: 76543 bytes. M**ax difference: 3264 bytes**
- Size of the polymorphic instance
  - ~ k if ω –> ∞
  - Roughly proportional to the overhead incurred by the execution of the polymorphic instance

Conclusion

- deGoal: build runtime code generators (a.k.a *compilettes*)
  - Code specialisation on runtime data values
  - Code specialisation on characteristics of the hardware
  - Applicable to embedded systems constrained wrt computing power and memory resources
- Polymorphic runtime code generation
  - Generic software countermeasure
    - The generated code can exploit hardware characteristics available
    - Compatible with state-of-the-art software countermeasures
  - Variation of the observation in time *and* space
  - Experimental validation on AES, side channel: the security is increased by a factor of 2000

Thanks!