DE LA RECHERCHE À L'INDUSTRIE

cea

www.cea.fr

leti & list

Runtime Code Generation for Code Polymorphism

Workshop on Runtime Code Generation for Secured Embedded Devices

Damien Couroussé

2015-12-03

AGENCE NATIONALE DE LA RECHERCHE
ANR

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

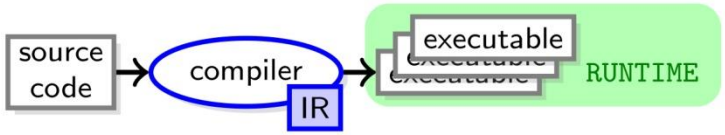
Outline

Tools for runtime code generation in embedded systems

- … for performance
- … for security

Approaches for code specialisation

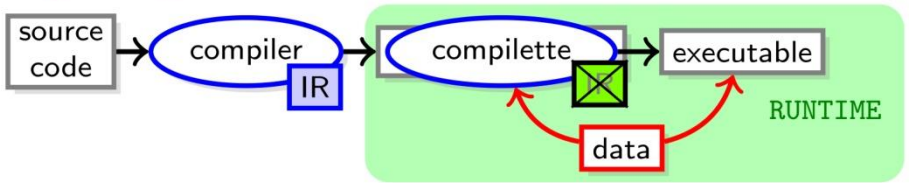
Static code versionning (e.g. C++ Templates)



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

Runtime code generation, with `deGoal`

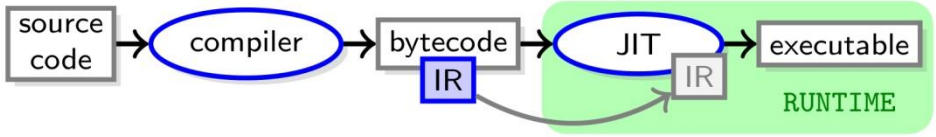
A *compilette* is an ad hoc code generator, targeting one executable



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

Dynamic compilation

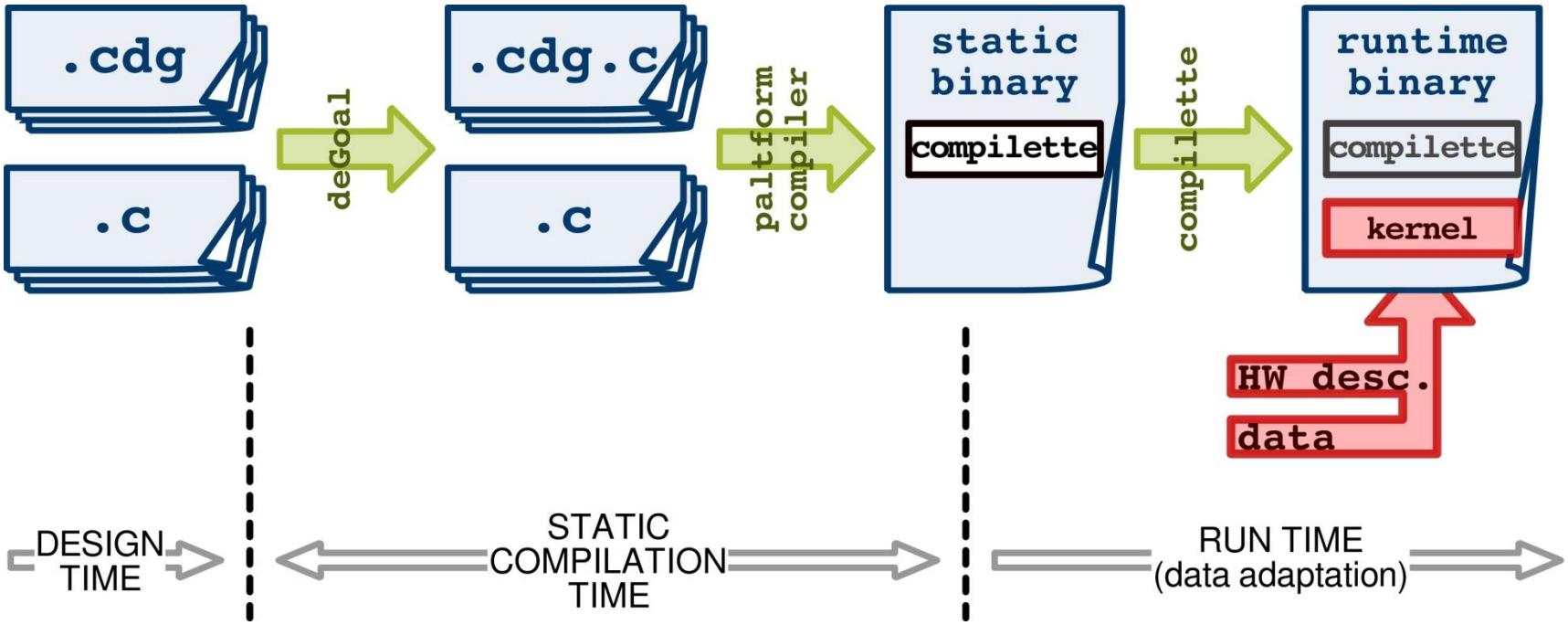
(JITs, e.g. Java Hotspot)



- 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

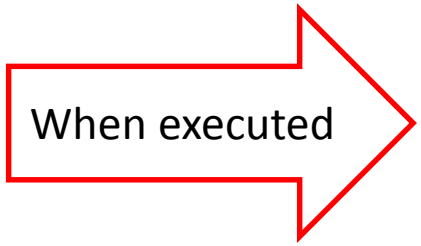
Example of deGoal code

- 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
]#
}
```



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

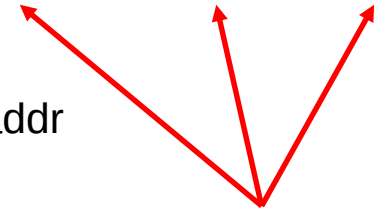
Example of deGoal code

- 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

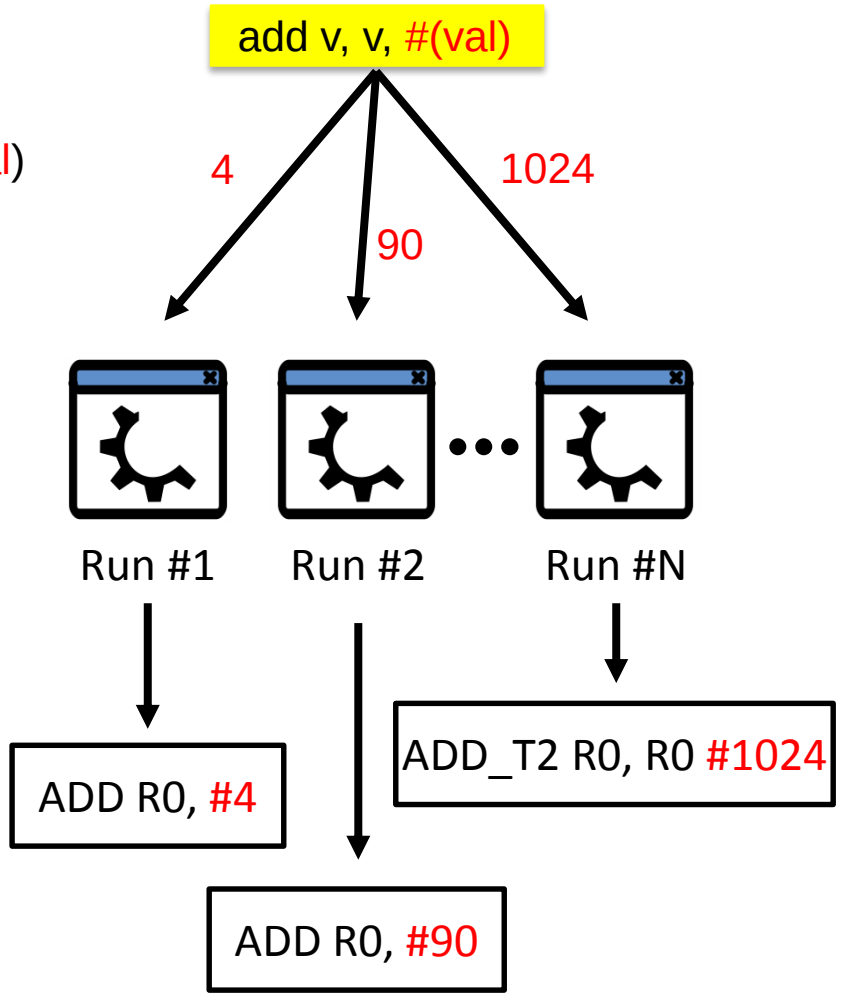
- Simple program example: vector addition

```
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

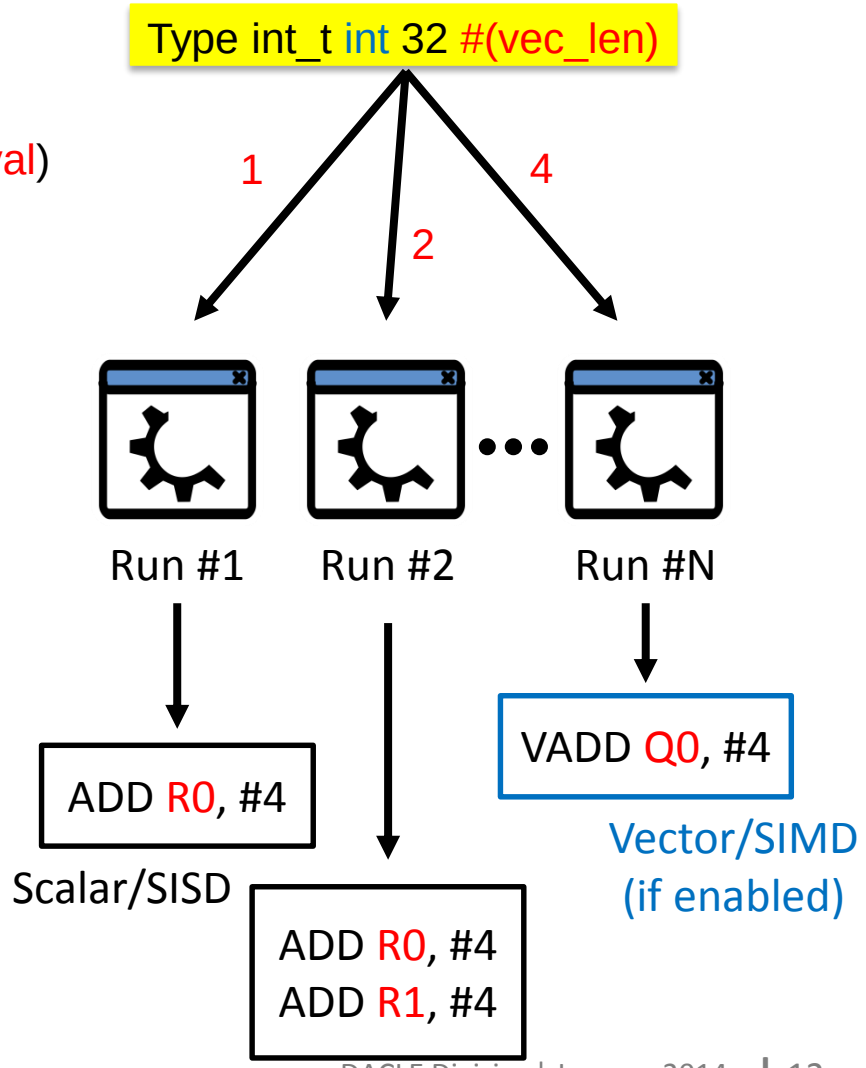
- Simple program example: vector addition

```
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
- …

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

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)

Countermeasures & polymorphism

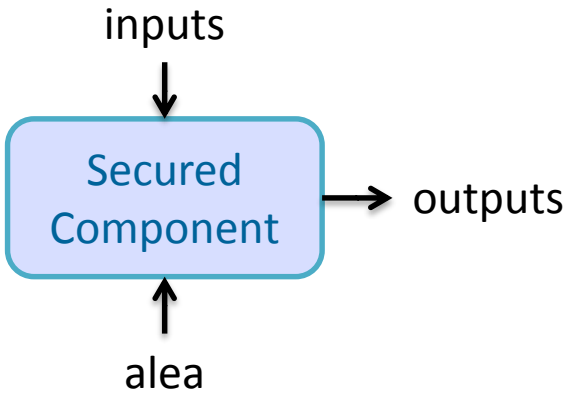
State of the Art

- Random register renaming [May 2011a, Agosta 2012]
- Out-of-Order execution :
  - At the instruction level [May 2011b, Bayrak 2012]
  - At the control flow level [Agosta 2014, Crane 2015]
- Execution of dummy instructions [Ambrose 2007, Coron 2009, Coron 2010]
- A few proof-of-concept implementations, not suitable for embedded devices [Amarilli 2011, Amarilli 2011, Agosta 2012]

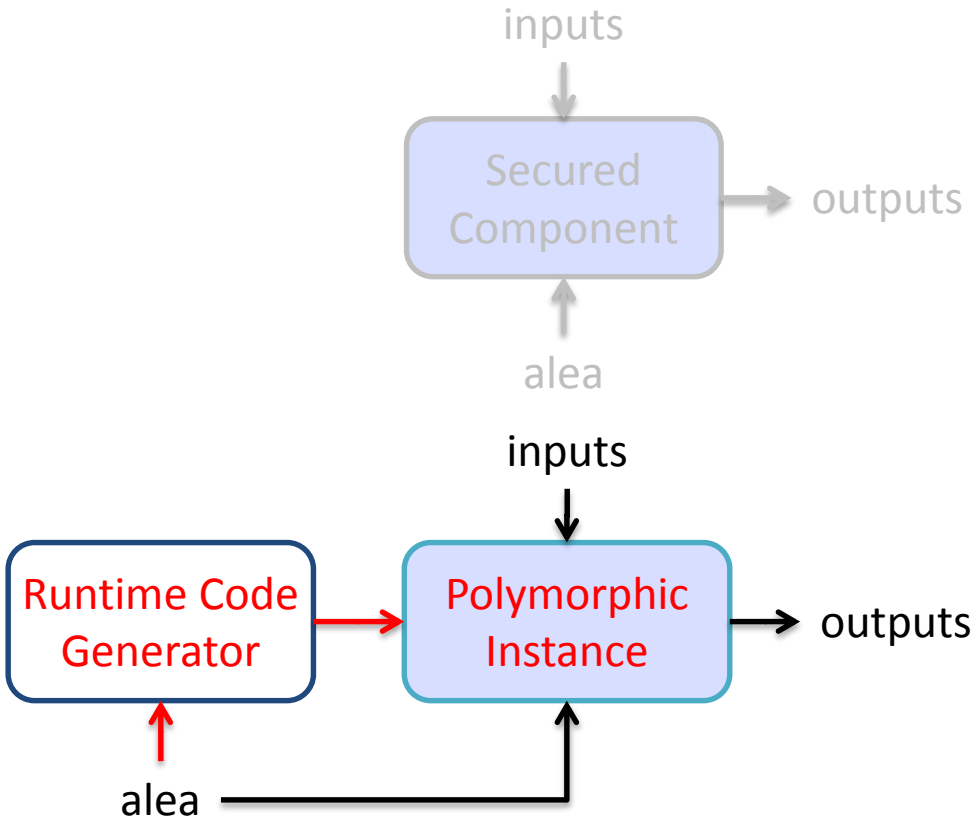
Our approach

- Pure software → portability, genericity
- Combination of *all* the polymorphic levers found in the state of the art
- Modest overhead (execution time & memory footprint)
- With runtime code generation

Polymorphic runtime code generation



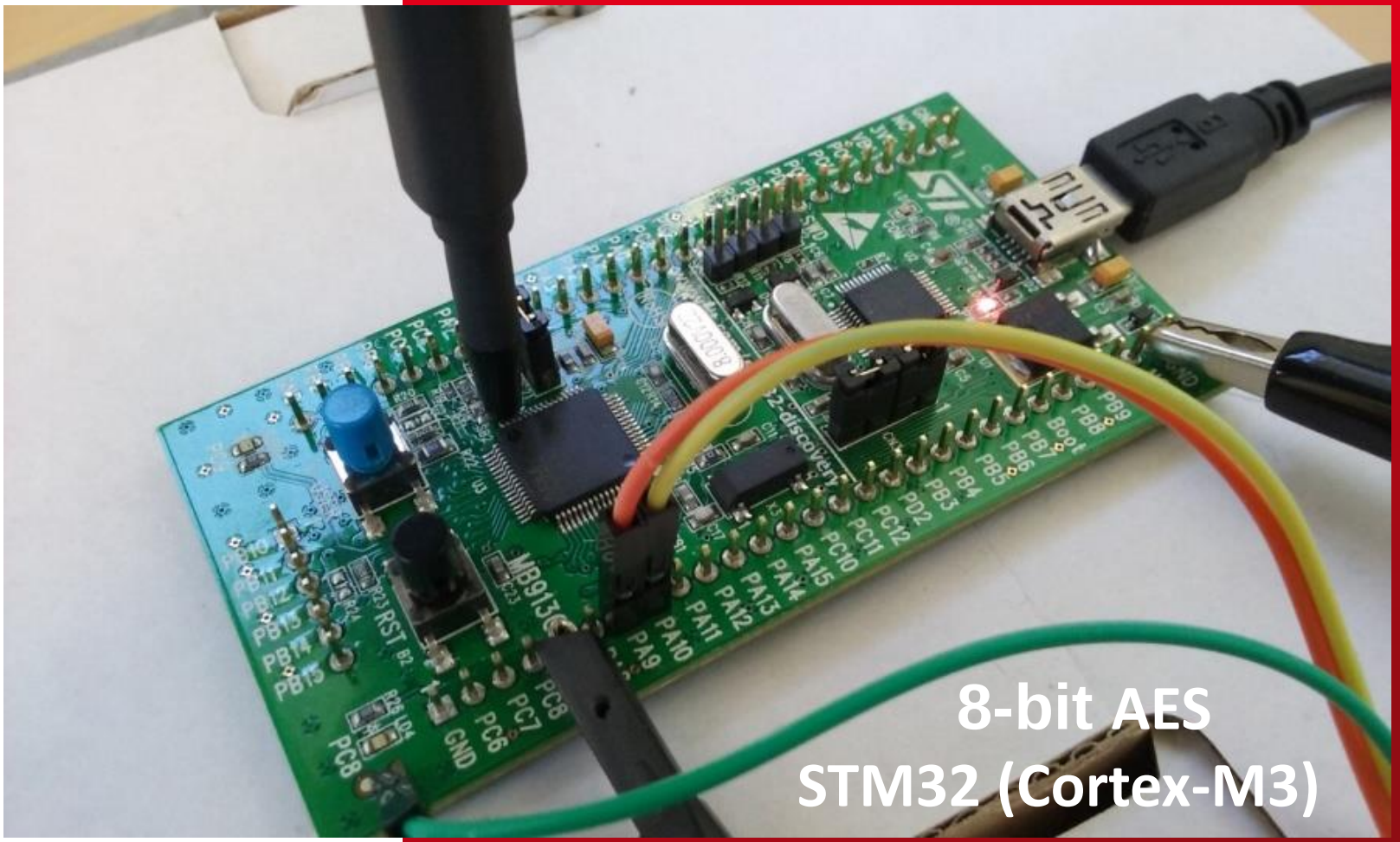
Polymorphic runtime code generation



Opportunities for polymorphic code generation

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

Demo



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

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

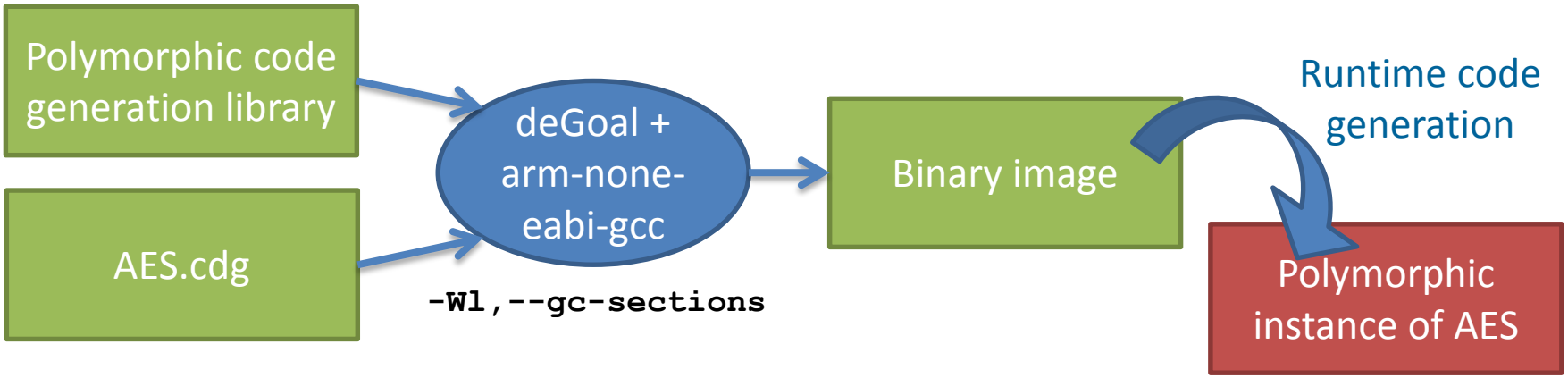
- Reorder instructions, but do not break the semantics of the code!
  - Defs
  - Uses
  - *Do not* take into account result latency and issue latency
  - Special treatments for… special instructions. E.g. branch instructions

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:
  - nop
  - Arithmetic (add, xor…)
  - Memory accesses (lw, lb, …)
  - Power hungry ones (mul, mac…)

AES 8-bit. 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. **Max difference: 3264 bytes**
- Size of the polymorphic instance
  - Size increase ~ ×1.30
  - Variable overhead. Highly depends on the code generation settings

AES 8-bit. Performance overhead

	Execution times (cycles)		
	Min	Max	Avg.
Reference AES	6385	6385	6385
Code generator	5671	12910	9345
Polymorphic instance	7185	9745	8303

	Execution time overhead	
Interval of code generation	COGITO	[Agosta, 2012] (× : *extrapolation*)
1	2,76	× *398*
5	1,59	× *80,4*
20	1,37	× *8,94*
100	1,31	5,00
2000	1,30	1,27
11600	1,30	1,10

Evaluation platform: ARM Cortex-M3, arm-none-eabi-gcc v4.8.1

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

INSTITUT CARNOT CEA LETI

INSTITUT CARNOT CEA LIST

MINATEC®

digiteo

leti

Centre de Grenoble
17 rue des Martyrs
38054 Grenoble Cedex

list

Centre de Saclay
Nano-Innov PC 172
91191 Gif sur Yvette Cedex

Bibliographic references

[Agosta 2012] Agosta, G., Barenghi, A., Pelosi, G.: A code morphing methodology to automate power analysis countermeasures. In: DAC. pp. 77–82. ACM (2012)

[Agosta 2014] Agosta, G., Barenghi, A., Pelosi, G., and Scandale, M.. 2014. A Multiple Equivalent Execution Trace Approach to Secure Cryptographic Embedded Software. DAC 2014.

[Ambrose 2007] Ambrose, Jude Angelo, Roshan G. Ragel, and Sri Parameswaran. "A smart random code injection to mask power analysis based side channel attacks." Hardware/Software Codesign and System Synthesis (CODES+ ISSS), 2007 5th IEEE/ACM/IFIP International Conference on. IEEE, 2007.

[Amarilli 2011] Amarilli, A., Müller, S., Naccache, D., Page, D., Rauzy, P., Tunstall, M.: Can Code Polymorphism Limit Information Leakage? In: WISTP (2011)

[Bayrak 2012] Bayrak, A.G., Velickovic, N., Ienne, P., Burleson, W.: An architecture-independent instruction shuffler to protect against side-channel attacks. TACO (2012)

[Coron 2009] Coron, J. S., & Kizhvatov, I. (2009). An efficient method for random delay generation in embedded software. In Cryptographic Hardware and Embedded Systems-CHES 2009 (pp. 156-170). Springer Berlin Heidelberg.

[Coron 2010] Coron, J. S., & Kizhvatov, I. (2010). Analysis and improvement of the random delay countermeasure of CHES 2009. In Cryptographic Hardware and Embedded Systems, CHES 2010 (pp. 95-109). Springer Berlin Heidelberg.

[Crane 2015] Crane, S., Homescu, A., Brunthaler, S., Larsen, P., & Franz, M. (2015). Thwarting cache side-channel attacks through dynamic software diversity. In Network And Distributed System Security Symposium, NDSS (Vol. 15).

[May 2001a] May, D., Muller, H., Smart, N.: Random register renaming to foil DPA. In: CHES, vol. LNCS 2162, pp. 28–38. Springer (2001)

[May 2001b] May, D., Muller, H.L., Smart, N.P.: Non-deterministic processors. In: ACISP'01. pp. 115–129. Springer (2001)

[Morpho 2013] Protection of Applets against hidden-channel analyses. US Patent 2013/0312110 A1