Runtime Code Polymorphism as a Protection against Physical Attacks

Damien Couroussé, Bruno Robisson, Thierno Barry, P Jaillon, Olivier Potin

FROM RESEARCH TO INDUSTRY
cea tech
leti & list

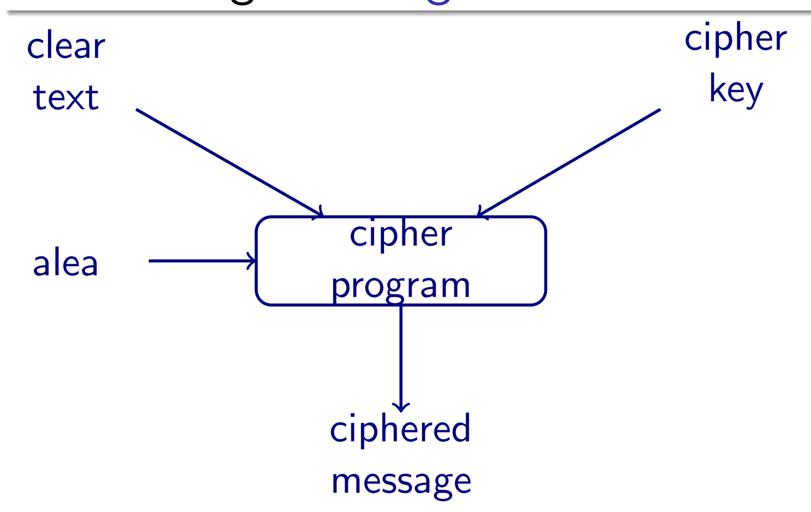
Runtime Code Generation to Secure Devices

D. Couroussé[1] B. Robisson[2] J.L. Lanet[3] T. Barry[1] P. Jaillon[4] O. Potin[4]
Univ. Grenoble Alpes, F-38000 Grenoble, France (1)
CEA, LIST, MINATEC Campus, F-38054 Grenoble, France (1)
CEA-Tech DPACA, Gardanne, France (2)
INRIA Rennes, France (3)
Ecole Nationale Supérieure des Mines de Saint-Etienne (ENSM.SE) Saint-Etienne, France (4)

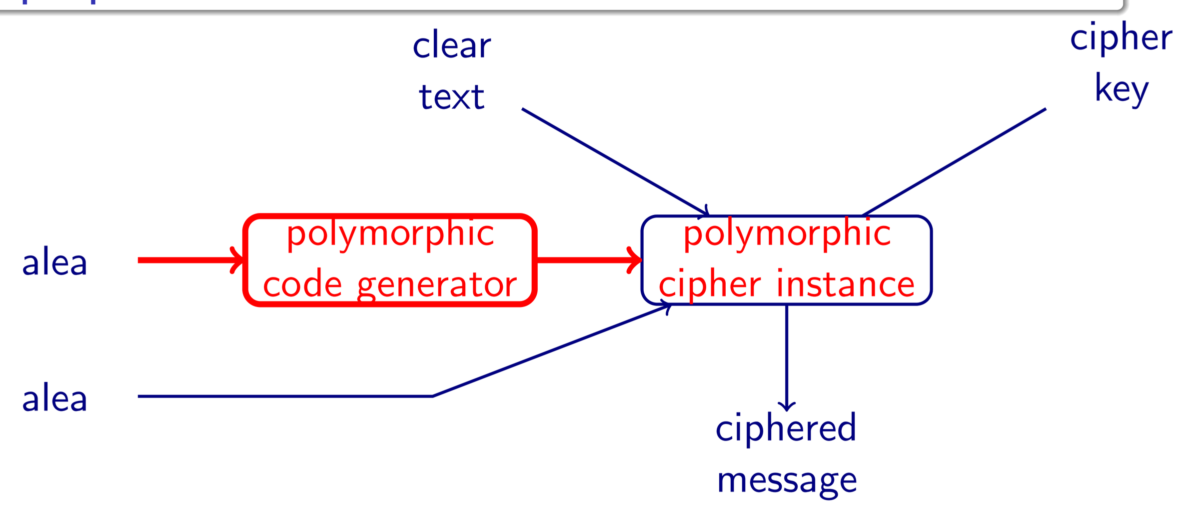
Core Idea: Runtime Code Polymorphism

Definition

Regularly changing the behaviour of a (secured) component, at runtime, while maintaining unchanged its functional properties



Default cipher scheme (static)



With runtime code polymorphism

Definition

- Regularly changing the behaviour of a (secured) component, at runtime, while maintaining unchanged its *functional properties*.

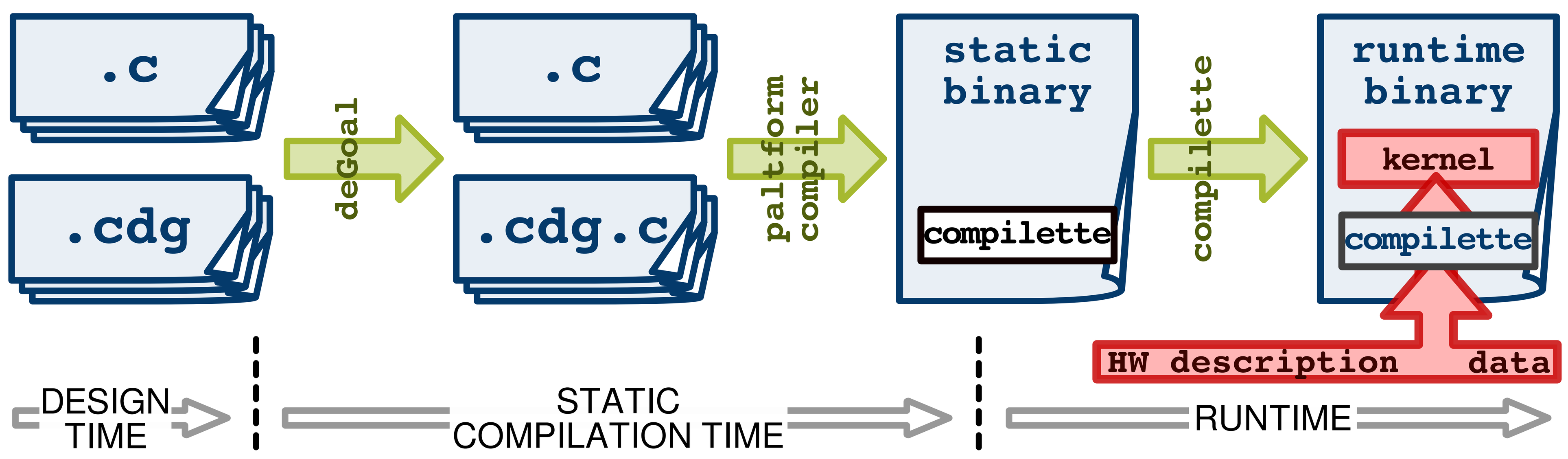
What for?

- Protection against reverse engineering of SW
  - the secured code is not available before runtime
  - the secured code regularly changes its form (code generation interval $\omega \geqslant 1$)
- 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

How?

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

Compilettes & deGoal in a Nutshell



Aim

- Modify kernel's binary instructions
- according to the input data
- whenever needed at runtime

The deGoal framework builds compilettes

A compilette is:

- an *ad hoc* code generator that targets *one* kernel
- aimed to be invocated at runtime

Polymorphic Code Generation

deGoal runtime capabilities

Performed *in this order*:

1. register selection
2. instruction selection
3. instruction scheduling

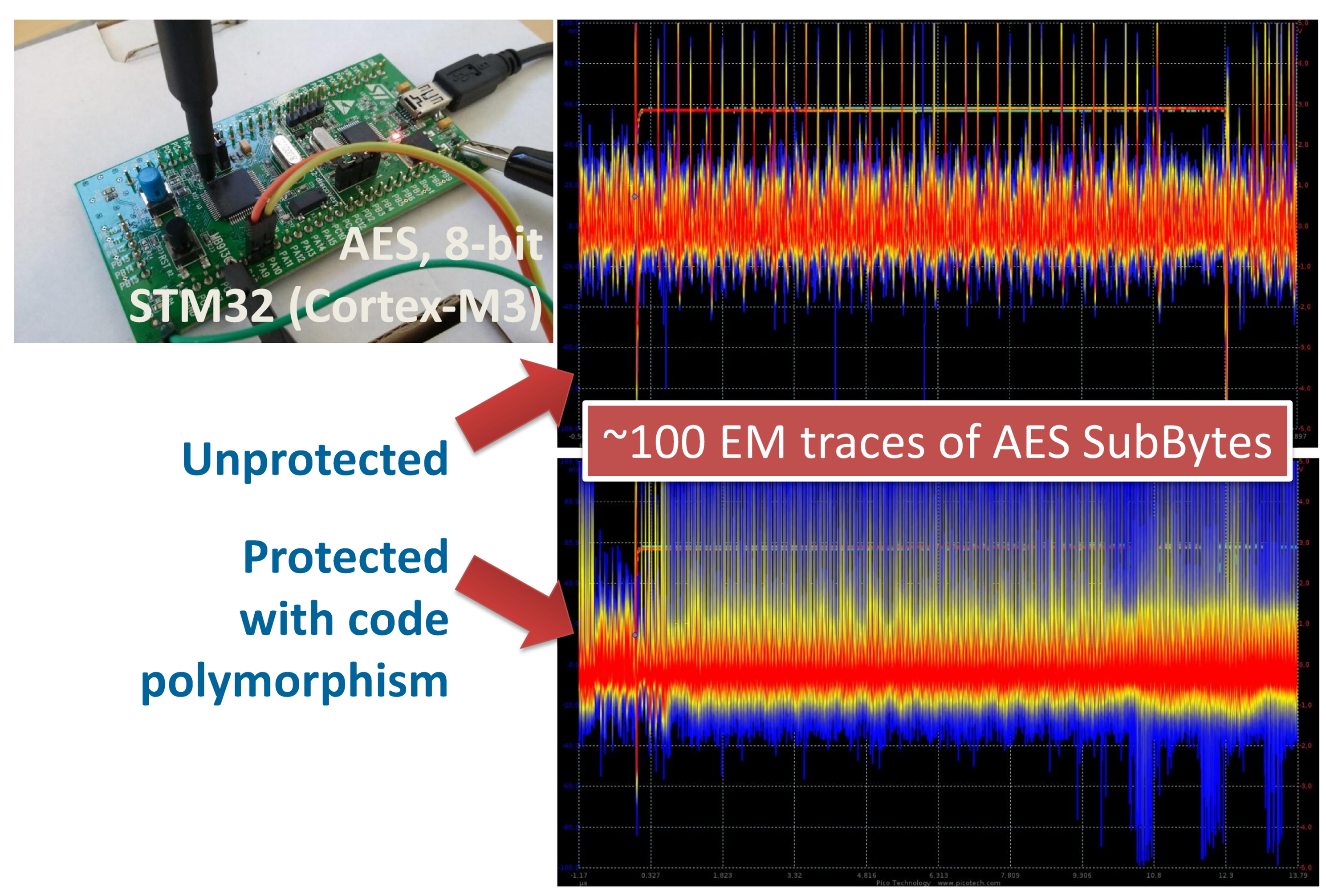
Adaptation to achieve runtime code polymorphism:

- Portability to very small processors and secure elements
  - Limited memory consumption
  - Fast runtime code generation
- Ability to combine with hardware countermeasures
- Introduce alea during runtime code generation [1,2,3]
- Polymorphism:
  - random mapping to physical registers [1]
  - use of semantic equivalences [2]
  - instruction scheduling [3]
  - insertion of dummy operations [3]

Example: polymorphic AES

Polymorphic implementation of the SubBytes function:

```
void gen_subBytes( cdg_insn_t* code
                 , uint8_t* sbox_addr
                 , uint8_t* state_addr)
{
#[
  Begin code Prelude
  Type uint32 int 32
  Alloc uint32 state, sbox, i, x, y
  mv state, #(state_addr)
  mv sbox, #(sbox_addr)
  mv i, #(0)
  loop:
    lb x, @(state+i)   //  x := state[i]
    lb y, @(sbox+x)    //  y := sbox[x]
    sb @(state+i), y   //  state[i] := y
    add i, i, #(1)
    bneq loop, i, #(16)
  rtn
  End
]#;
}
```



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

References

Overview of our approach for runtime code generation with compilettes:
H.-P. Charles, D. Couroussé, V. Lomüller, F. A. Endo, and R. Gauguey, "deGoal a Tool to Embed Dynamic Code Generators into Applications," in Compiler Construction, 2014, vol. 8409.

Runtime code generation for micro-controllers with less than 1kB RAM:
C. Aracil and D. Couroussé, "Software Acceleration of Floating-Point Multiplication using Runtime Code Generation," in Proceedings of the 4th International Conference on Energy Aware Computing, 2013.

Instruction scheduling for VLIW processors:
D. Couroussé, V. Lomüller, and H.-P. Charles, Introduction to Dynamic Code Generation – an Experiment with Matrix Multiplication for the STHORM Platform. Springer Verlag, 2013, pp. 103–124.