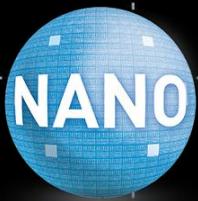


SECURING EMBEDDED SOFTWARE WITH COMPILERS



Damien Couroussé | CEA / LIST / DACLE
CEA-LETI LID, Minatec Grenoble, 2019-06-28

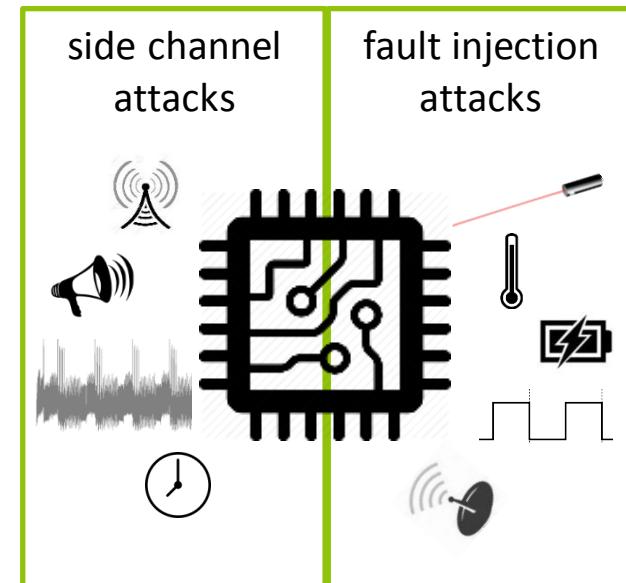
PHYSICAL ATTACKS

A major threat against secure embedded systems

- The most effective attacks against implementations of cryptography
- Relevant against many parts of CPS/IoT: bootloaders, firmware upgrade, etc.
- Recently used to leverage software vulnerabilities [1]

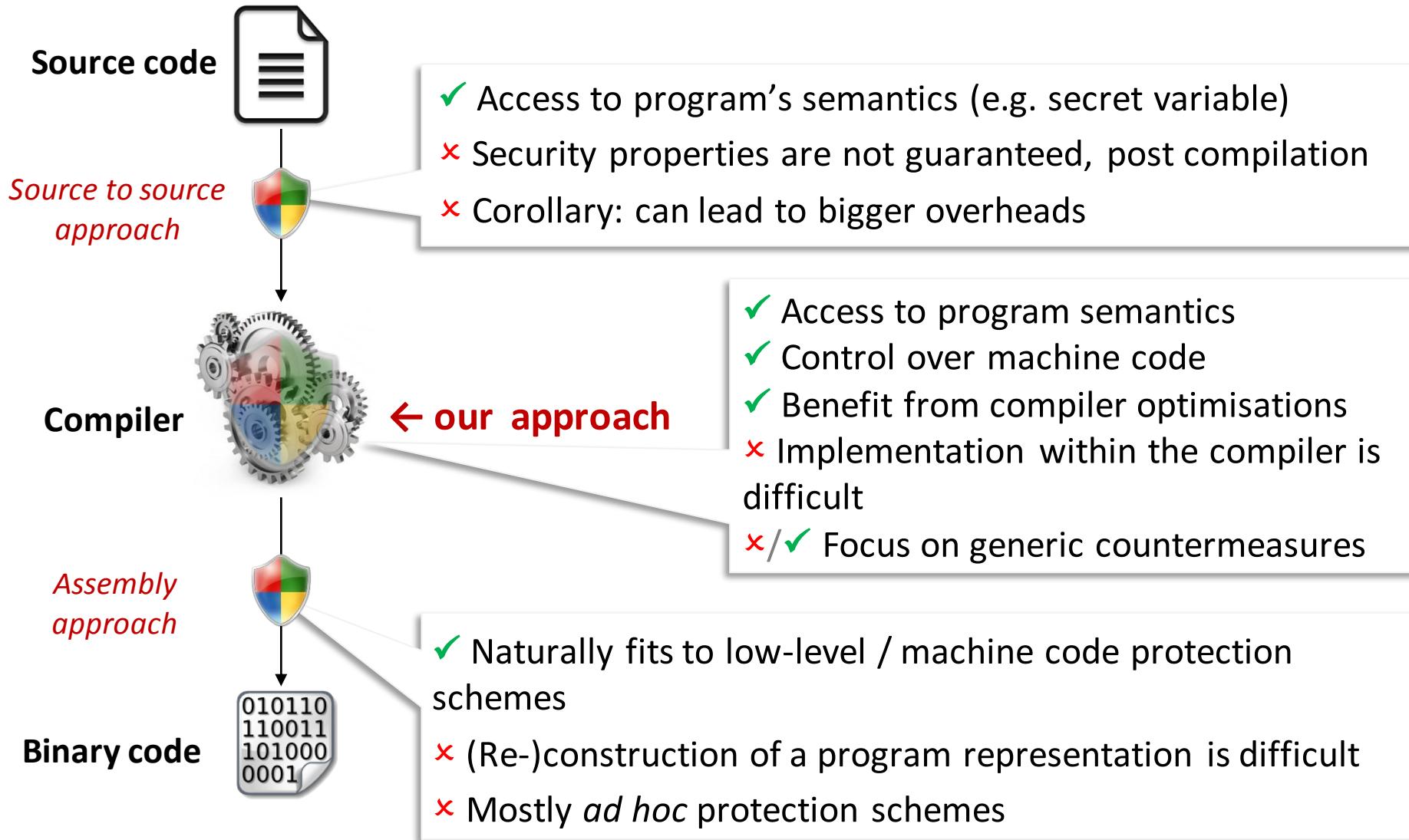
In practice,

- An attacker mostly uses logical attacks if the target is unprotected (e.g. typical IoT devices): buffer overflows, ROP, protocol vulnerabilities, etc.
- All high security products embed countermeasures against side-channel and fault injection attacks. E.g. Smart Cards, payTV, military-grade devices.
 - Using a combination of hardware *and* software countermeasures
- Tools for Side-channel and fault injection are getting really affordable



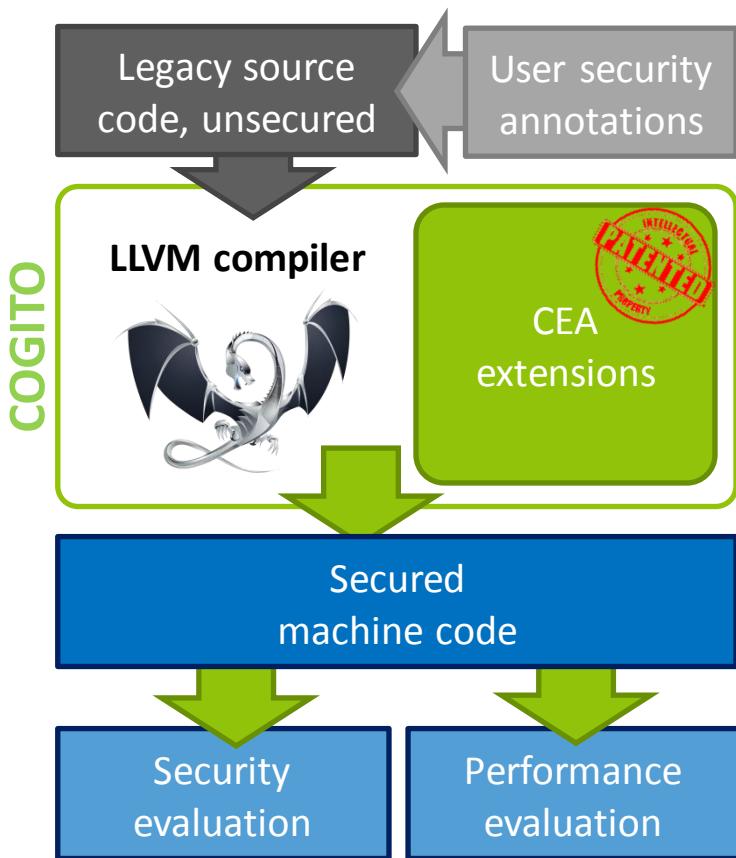
[1] A. Cui and R. Housley, 'BADFET: Defeating Modern Secure Boot Using Second-Order Pulsed Electromagnetic Fault Injection', presented at the WOOT, 2017.

AUTOMATED APPLICATION OF COUNTERMEASURES WITH A COMPILER



Automated application of software countermeasures against physical attacks

→ A toolchain for the compilation of secured programs



Several countermeasures

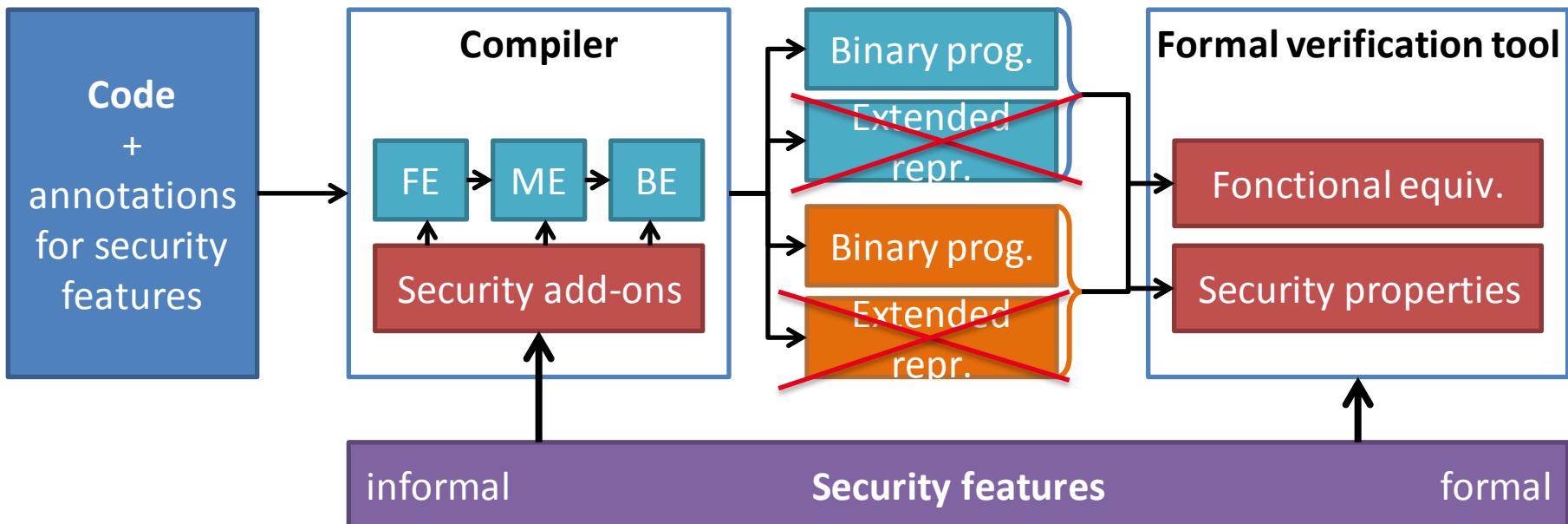
- **Fault tolerance**, including multiple fault injections
- **Execution Integrity & Control-Flow Integrity**
 - Detection of perturbations on the instruction path, at the granularity of a single machine instruction
- **Side channel hiding**

Tools for **security and performance evaluations**

Based on **LLVM**: an industry-grade, state-of-the art compiler (competitive with GCC)

SECURING AND VERIFYING PROGRAMS

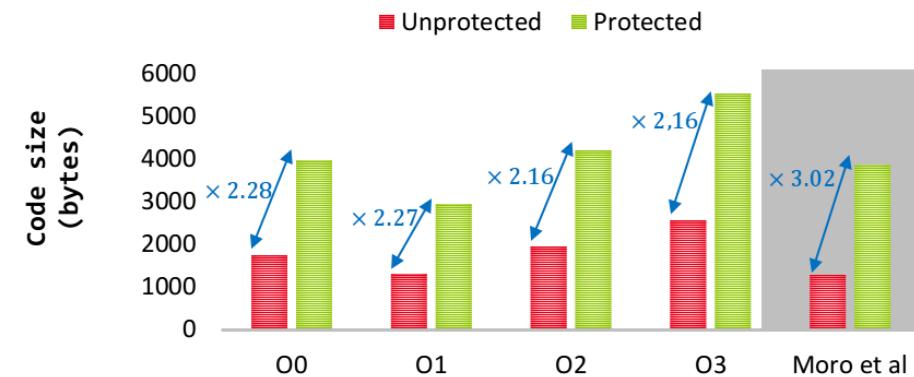
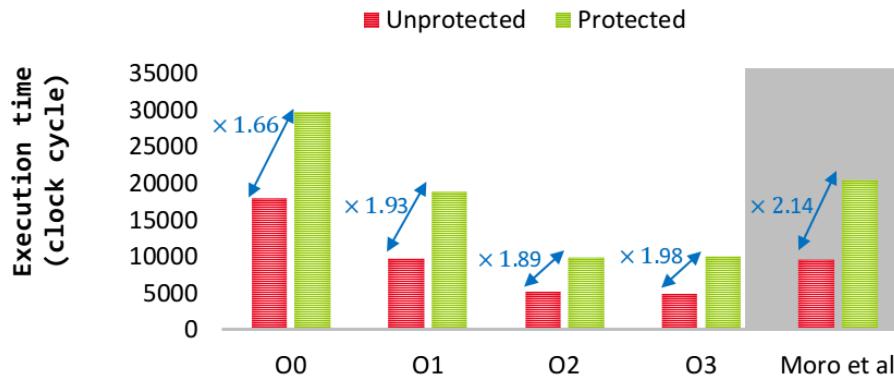
- **Compilation:** automation of the application of software countermeasures against fault attacks and side-channel attacks
- **Functional verification:** of the secured machine code (equivalence with an unprotected version of the same program)
- **Security verification:** correctness of the applied countermeasures w.r.t a security model



On-going joint work with LIP6, Paris (PROSECCO – ANR 2015)

Objective: the program is not perturbed by the injection of faults

- Countermeasure based on a protection scheme **formally verified for the ARM architecture** [Moro et al., 2014, Barry et al. 2016]
- **Automatic application** by the compiler
- Allow to **parameterize** level of protection
- **Generalisation** of [Moro et al., 2014] to **multiple faults of configurable width**
- Target: ARM Cortex-M cores



- **Fine-grained countermeasure** applied to critical functions **reduces the execution overhead** below x1.23 and size overheads below x1.12 [Barrys' thesis, 2017]

[Moro et al., 2014] Moro, N., Heydemann, K., Encrenaz, E., & Robisson, B. (2014). Formal verification of a software countermeasure against instruction skip attacks. *Journal of Cryptographic Engineering*, 4(3), 145-156.

[Barry et al. 2016] Barry, T., Couroussé, D., & Robisson, B. (2016, January). Compilation of a Countermeasure Against Instruction-Skip Fault Attacks. In Proceedings of the Third Workshop on Cryptography and Security in Computing Systems (pp. 1-6). ACM.

Objective: monitoring program execution integrity, at runtime

Combined protections:

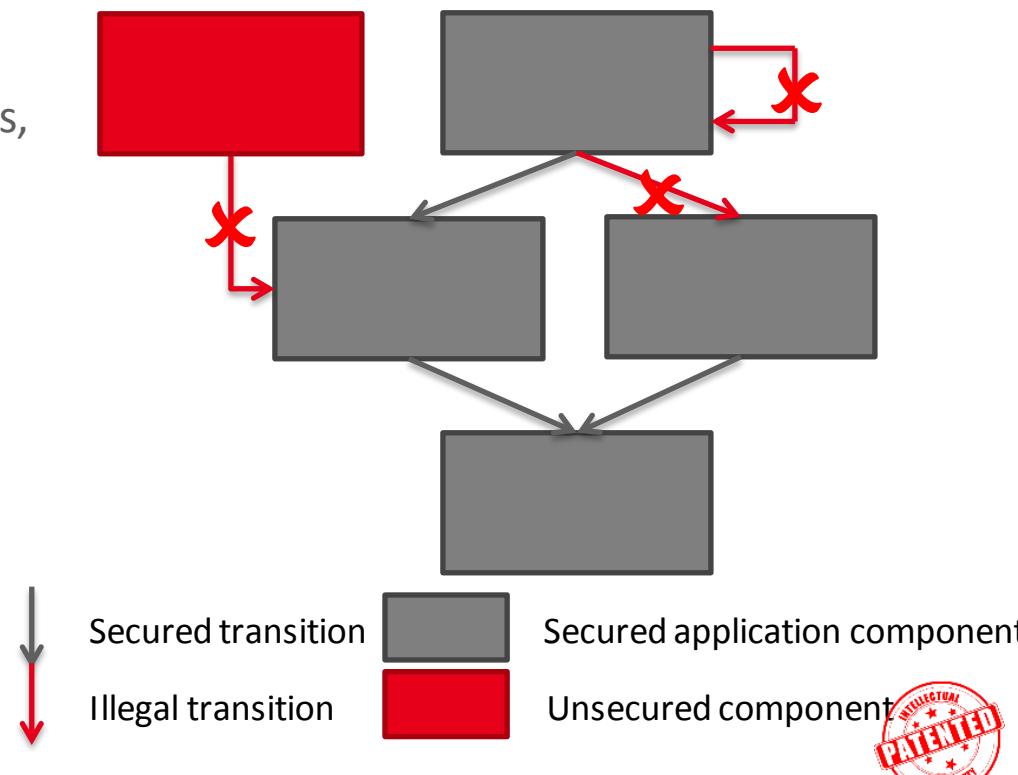
- Protection of the control-flow of an application (Control-Flow Integrity)
- Beyond CFI: protection of branchless sequences of instructions, at the granularity of a single machine instruction

Coverage:

- Alteration of the PC (instruction skips, branches)
- Corruption of branches
- Alteration of branch conditions

Two implementations

- Software only countermeasure.
Implementation for ARM
- HW-SW countermeasure. Fine-grain execution integrity, verification & authentication.

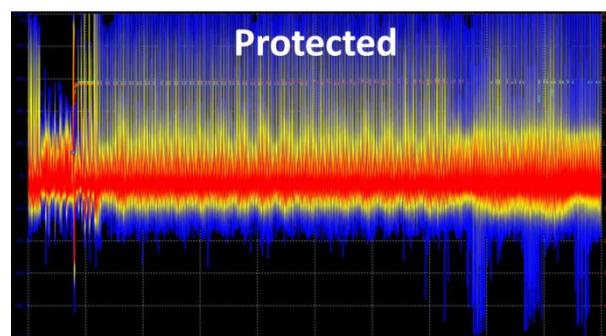


SIDE CHANNEL HIDING WITH CODE POLYMORPHISM

Code polymorphism: regularly changing the observable behavior of a program, at runtime, while maintaining unchanged its functional properties,

- Protection against physical attacks: side channel & fault attacks
 - Changes the spatial and temporal properties of the secured code
 - Can be combined with other state-of-the-Art HW & SW Countermeasures
- Can run on low-end embedded systems with only a few kB of memory
 - Illustrated below: STM32F1 microcontroller with 8kB of RAM

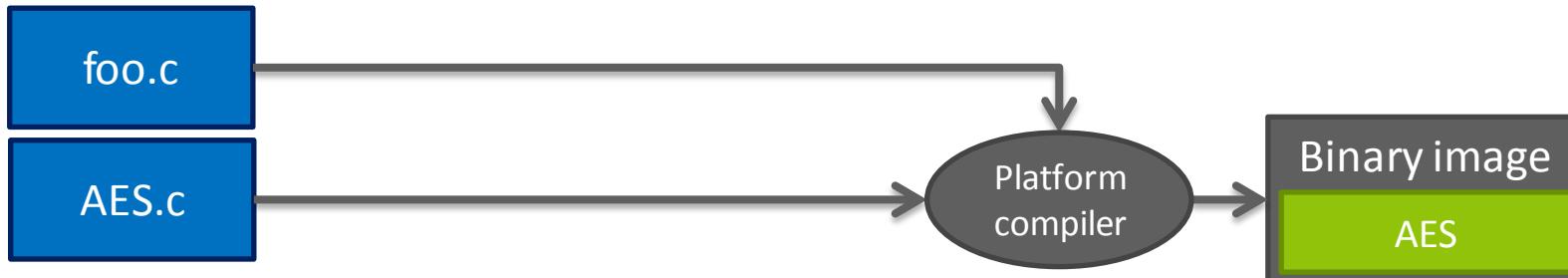
Compliant with certification standards (Common Criteria, CSPS, etc.)



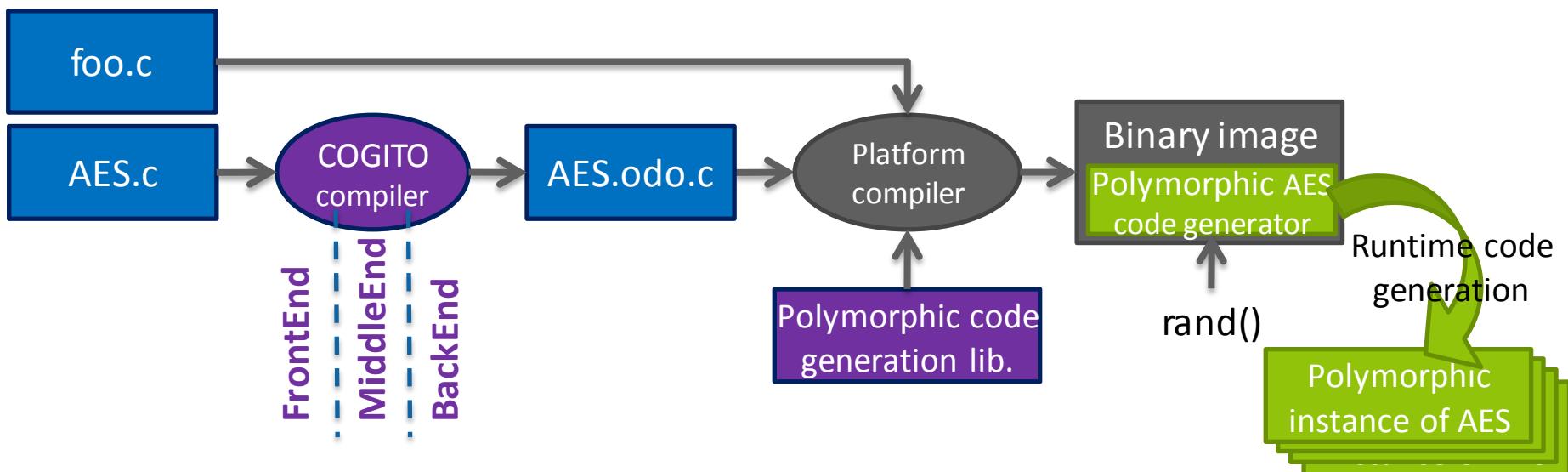
CODE POLYMORPHISM: WORKING PRINCIPLE

Runtime code generation for embedded systems

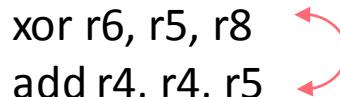
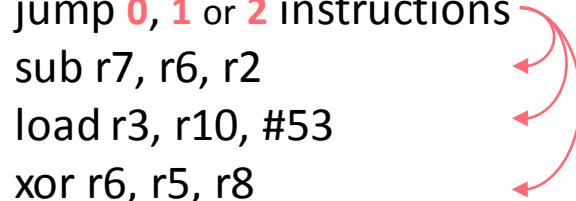
Reference version:



Polymorphic version, with COGITO:



CODE TRANSFORMATIONS USED AT RUNTIME

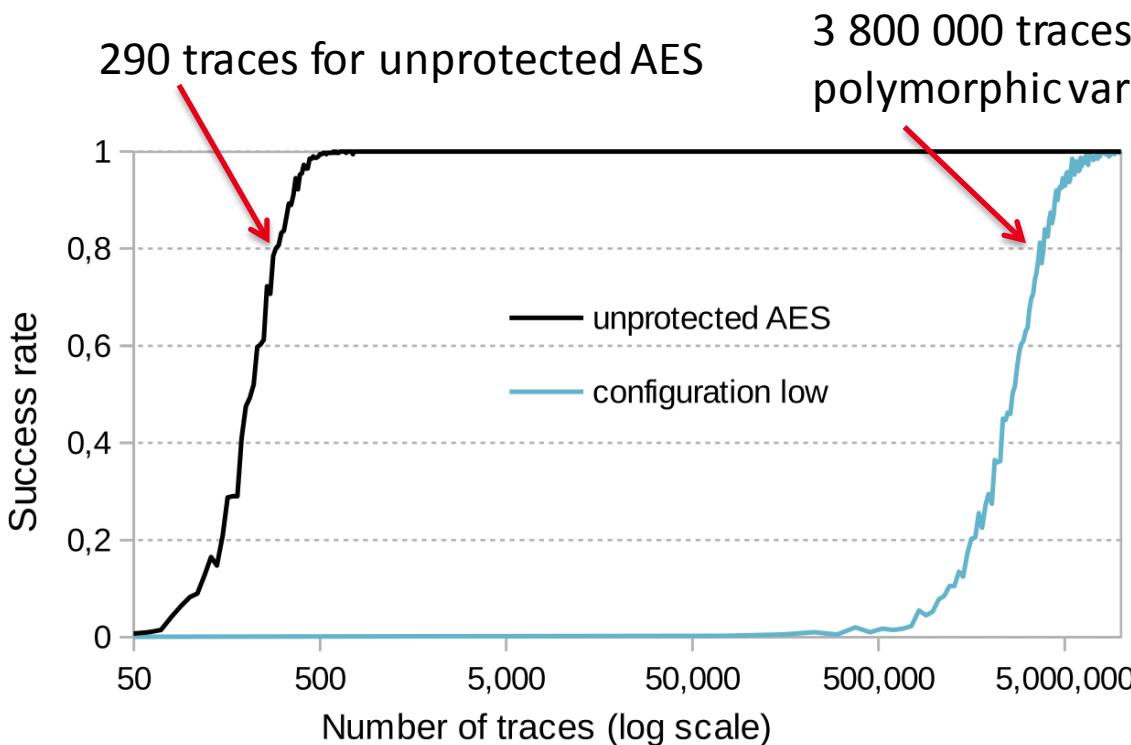
<pre>add r4, r4, r5 xor r6, r5, r8</pre>	<p>Register shuffling RANDOM general purpose register permutation</p>  <pre>add r11, r11, r7 xor r8, r7, r5</pre>	<p>Instruction shuffling independent instructions are emitted in a RANDOM order</p> 
<p>Semantic variants replacement of an instruction by a RANDOMLY selected semantic variant</p> <pre>add r4, r4, r5 xor r6, r5, #12348 xor r6, r6, r8 xor r6, r6, #12348</pre>	<p>Noise instructions insertion of a RANDOM number of RANDOMLY chosen noise instructions</p> <pre>add r4, r4, r5 sub r7, r6, r2 load r3, r10, #53 xor r6, r5, r8</pre> <p style="text-align: right;"><i>useless instructions</i></p>	<p>Dynamic noise RANDOM insertion of noise instructions with a RANDOM jump</p>  <pre>add r4, r4, r5 jump 0, 1 or 2 instructions sub r7, r6, r2 load r3, r10, #53 xor r6, r5, r8</pre>

TOOLCHAIN CONFIGURABILITY

Period of regeneration \mathbb{N} (or custom regeneration policies)	Register shuffling $\{0, 1\}$	Instruction shuffling $\{0, 1\}$
<p>Total configuration space: $\{0, 1\}^2 \times \{0, 1, 2\}^2 \times \mathbb{R} \times \mathbb{N}^3$</p>		
Semantic variants $\{0, 1, 2\}$	Noise instructions $\{0, 1, 2\} \times \mathbb{R} \times \mathbb{N}$	Dynamic noise \mathbb{N}
<p>A huge number of polymorphic variants:</p> <ul style="list-style-type: none"> • 10 original machine instructions $\rightarrow 6 \cdot 10^{42}$ variants • AES with 278 machine instructions $\rightarrow 10^{27}$ variants (pessimist bound) 		

A SECURITY EVALUATION

- Basis: polymorphic configuration with low variability
- Acquisition of traces from Electro-Magnetic observations
- CPA on SBOX 1st output with HW model
- Experimental platform not designed for security applications (hence the weak results on the unprotected version)



3 800 000 traces with low polymorphic variability

Experimental results

- This polymorphic version requires 13000x more traces
- Execution time overhead: x2.5 including generation cost

More results in
[TACO 2019]

AUTOMATED APPLICATION OF CODE POLYMORPHISM

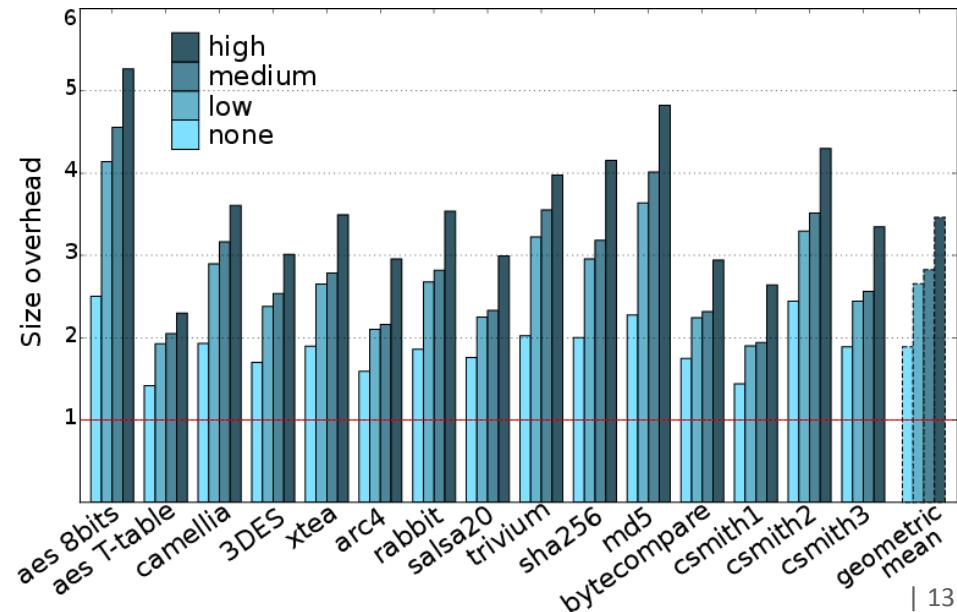
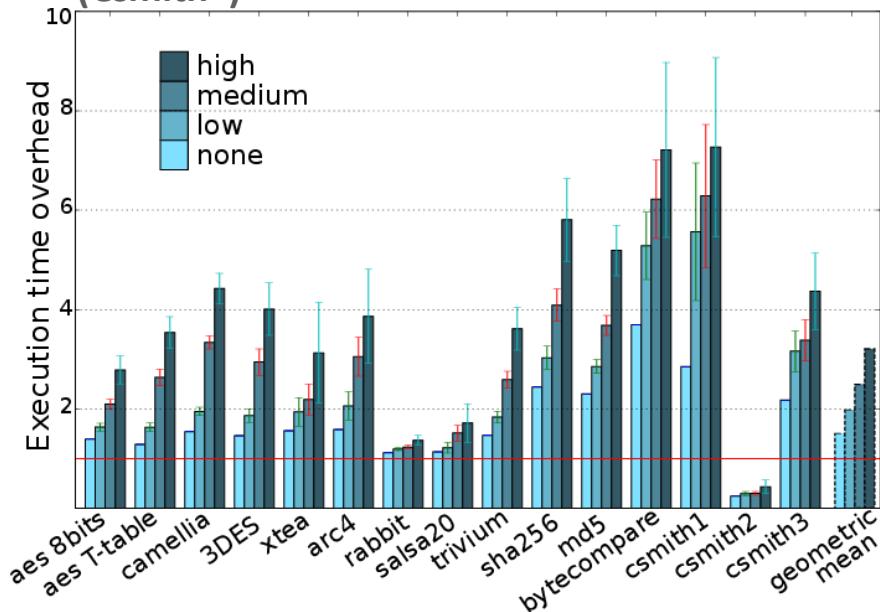
Declaration of polymorphism with a compiler option

- -polymorphic-function foo will compile function foo into a polymorphic implementation,
- -polymorphic will compile all functions found in the compiled source file into polymorphic implementations.

Many configurable levels of polymorphic transformations => security/performance tradeoff

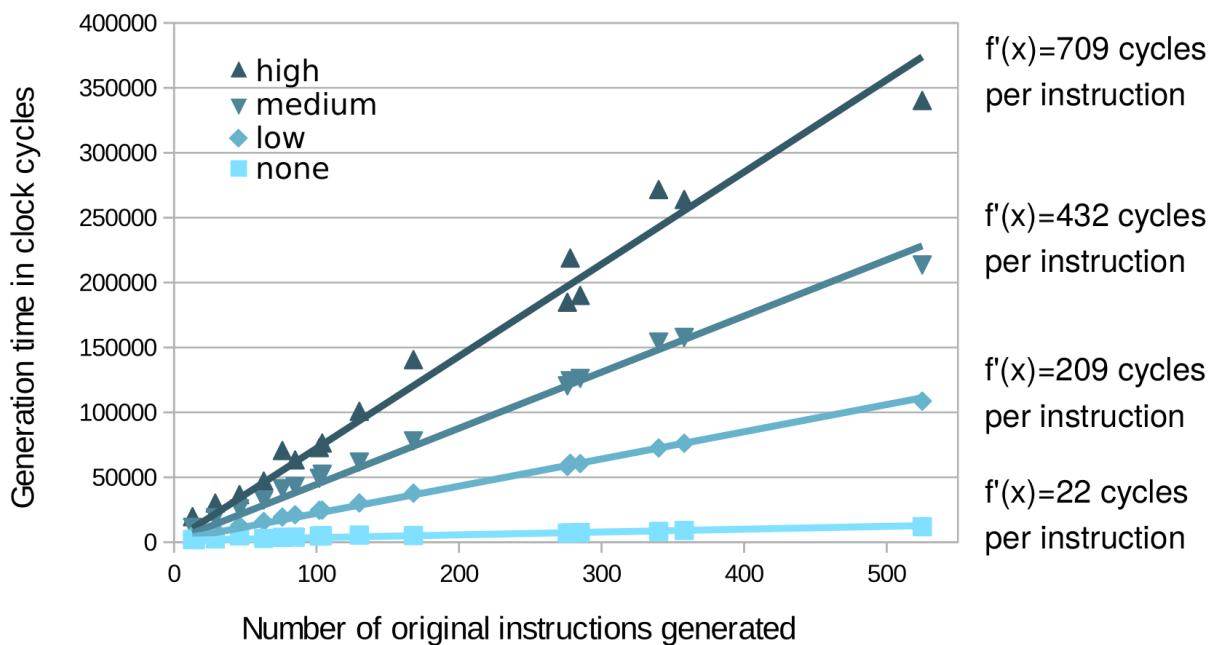
- Nature and parameters of the code transformations: random allocation of registers, semantic variants, instruction shuffling, insertion of noise instructions.
- Frequency and policy for runtime code regeneration
- Memory protections
- Leveraging OS-level features, e.g. concurrency

Components evaluated: ciphers, hash functions, simple authentication, random generated codes
(Csmith*)



PERFORMANCE EVALUATION OF RUNTIME CODE GENERATION

Configuration		Execution time overhead (geo. mean)	Size overhead (geo. mean)
none	■	x1.40	x1.70
low	◆	x2.31	x2.87
medium	▼	x2.45	x3.44
high	▲	x4.03	x3.81



$f(x) = 709$ cycles
per instruction

$f(x) = 432$ cycles
per instruction

$f(x) = 209$ cycles
per instruction

$f(x) = 22$ cycles
per instruction

Overheads depend
on configuration
→ trade-off to find

Runtime code
generation done in
linear complexity

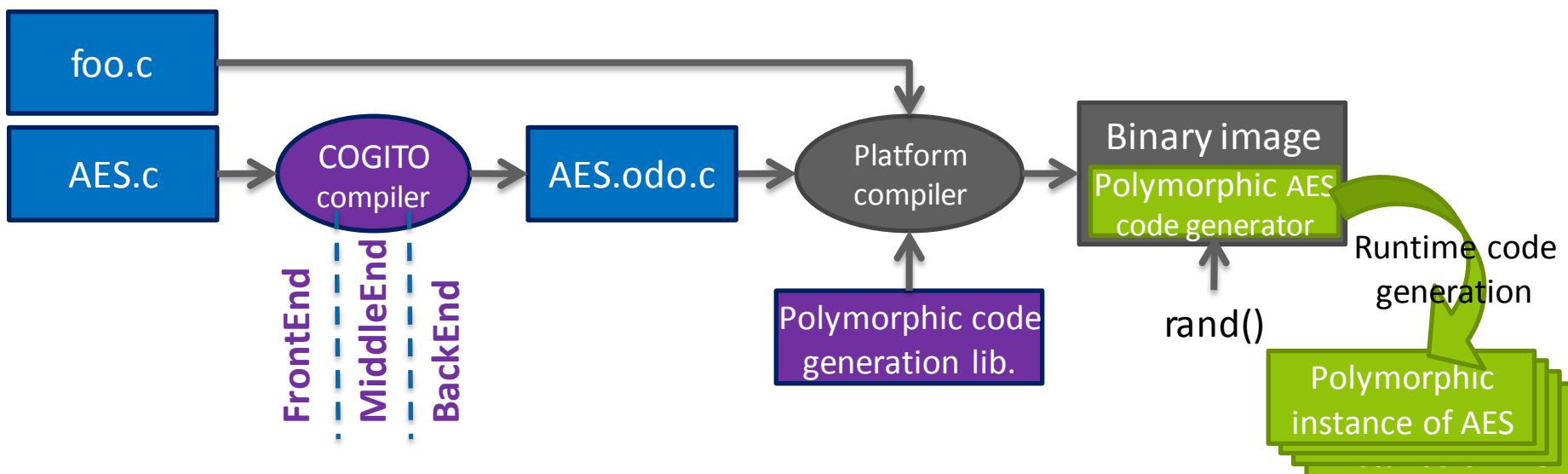
More results in
[TACO 2019]

CODE POLYMORPHISM: CHALLENGES

Bottlenecks for the use of runtime code generation in embedded systems:

- Memory allocation of code buffers
 - No Operating System (no malloc), no virtual memory.
- Management of memory permissions (read, write, execute)
 - Runtime code generation requires write access to program memory

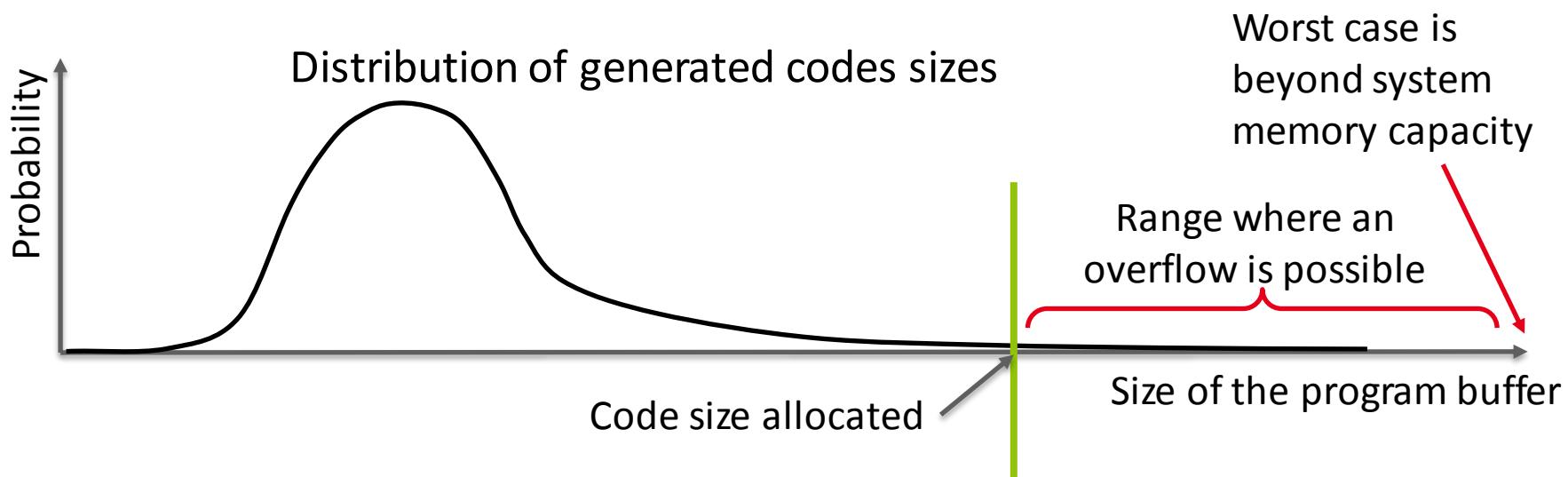
Polymorphic version, with COGITO:



MEMORY ALLOCATION OF CODE BUFFERS

Challenges

- No Operating System, no dynamic memory allocation (`malloc`), no MPU
- Generated code has a variable size
- Largest possible code size does not fit in system memory



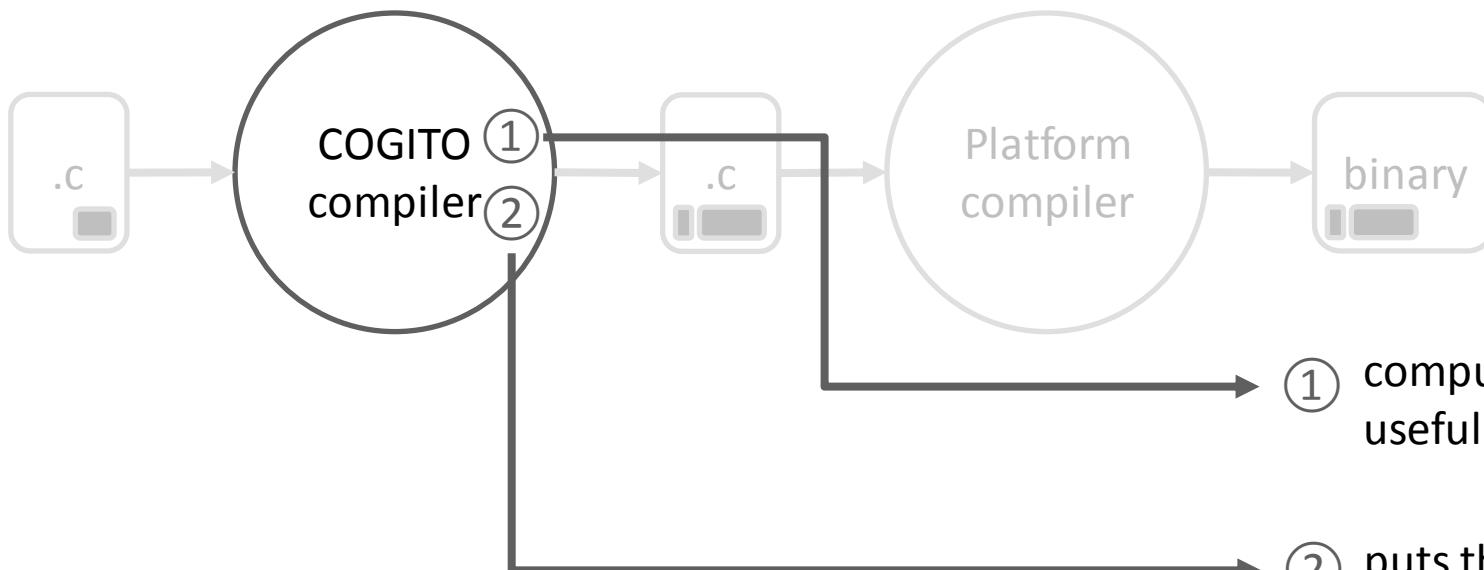
Idea: compute a realistic code size suitable for $(1-p)$ code generations.

- Threshold p : probability of memory overflow
- $p = 10^{-6}$ by default
- Computation of the code size done automatically by the compiler

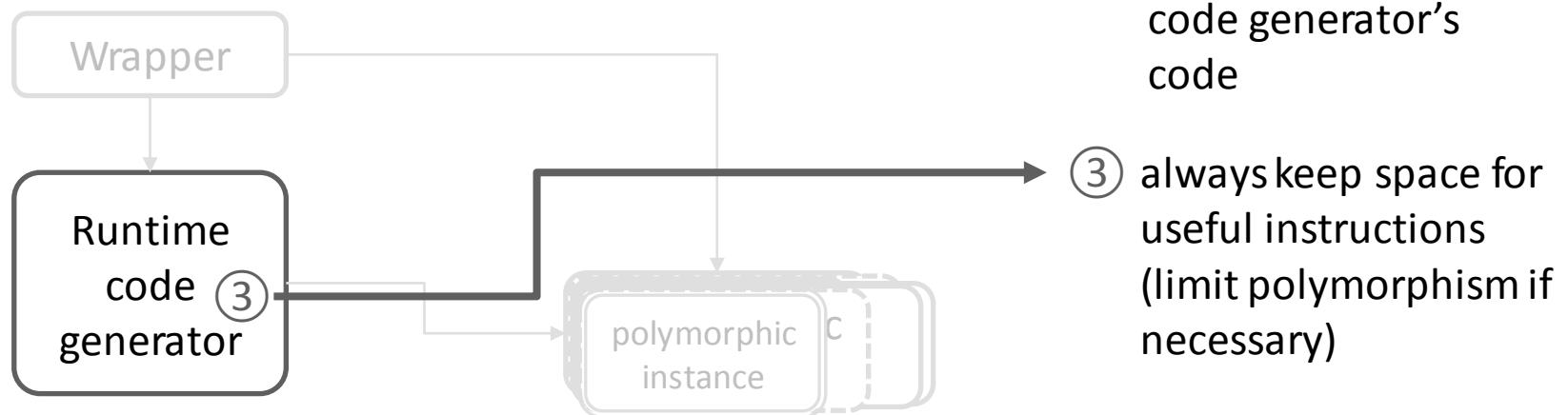
For a 100 instructions code (low config.), allocated size is **5x** smaller than worst case!

PREVENTION OF CODE BUFFER OVERFLOWS

STATICALLY

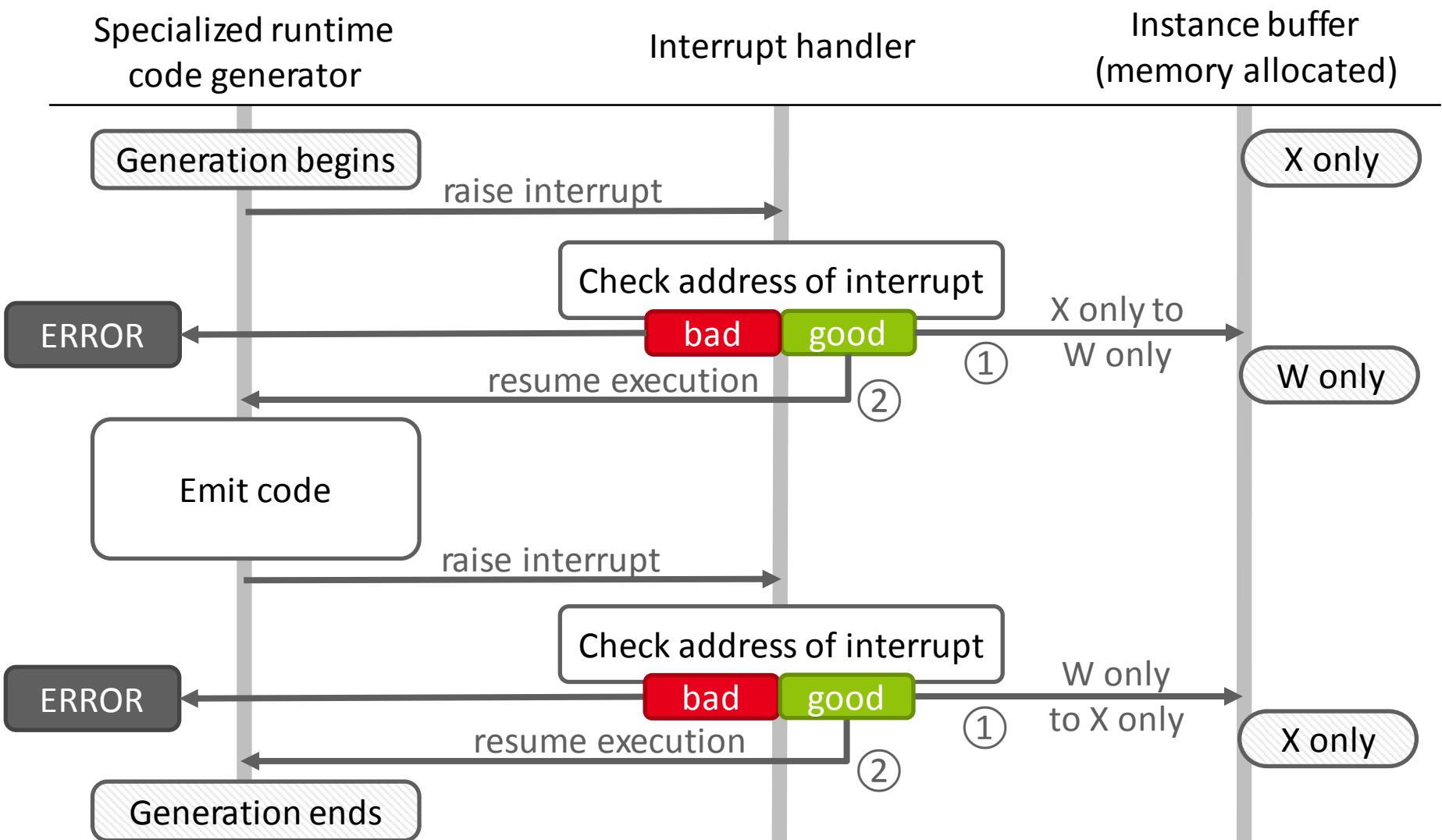


RUNTIME



MANAGEMENT OF MEMORY PERMISSIONS

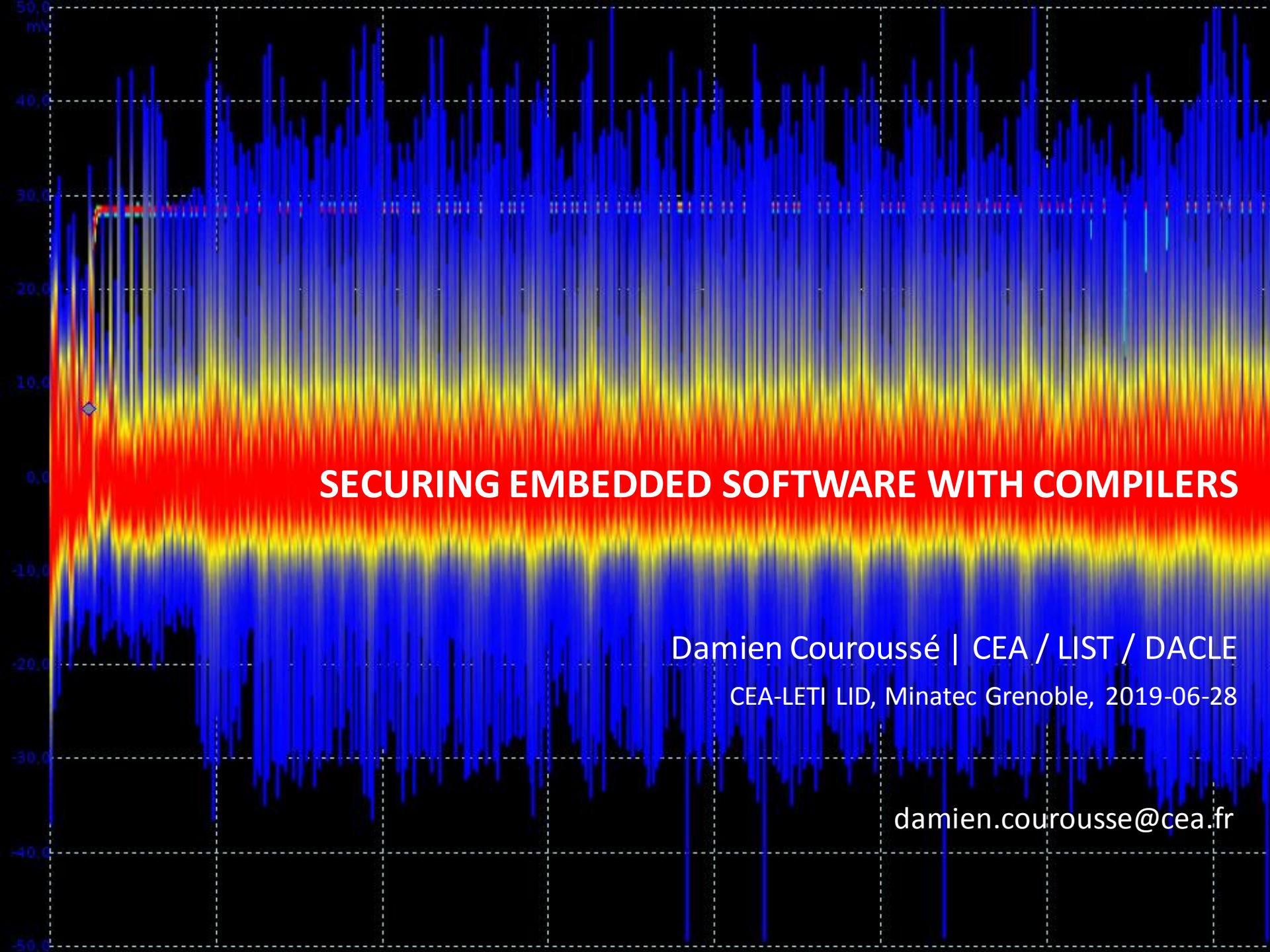
Objective: Guarantee $W \oplus X$ and that only the generator can write into the buffer



- **Leverage the compiler to implement counter-measures**
 - Automation, flexibility, configurability
- **Leverage compiler analysis and compiler optimisations to improve the effectiveness of counter-measures**

Ongoing directions

- **Hardware security with software-only counter-measures is impossible challenging**
 - Challenge your threat model
 - HW/SW co-design of countermeasures



SECURING EMBEDDED SOFTWARE WITH COMPILERS

Damien Couroussé | CEA / LIST / DACLE

CEA-LETI LID, Minatec Grenoble, 2019-06-28

damien.courousse@cea.fr