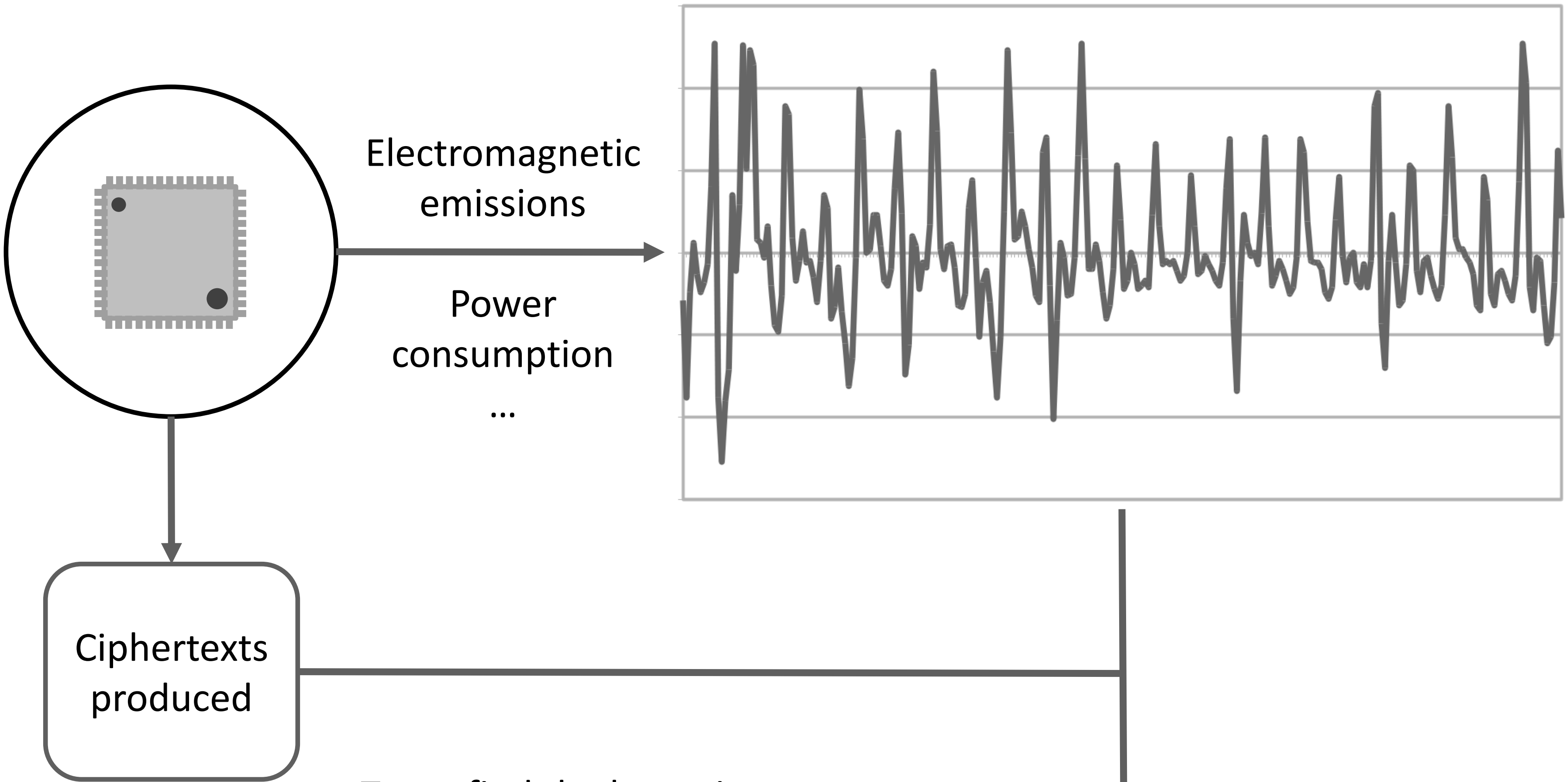
AUTOMATED SOFTWARE PROTECTION FOR THE MASSES AGAINST SIDE-CHANNEL ATTACKS

Nicolas Belleville [1]
Damien Couroussé [1]
Karine Heydemann [2]
Henri-Pierre Charles [1]

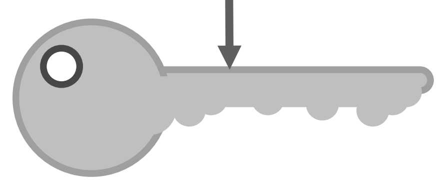
[1] Univ Grenoble Alpes, CEA, List, F-38000 Grenoble, France
firstname.lastname@cea.fr
[2] Sorbonne Université, CNRS, LIP6, F-75005, Paris, France
firstname.lastname@lip6.fr

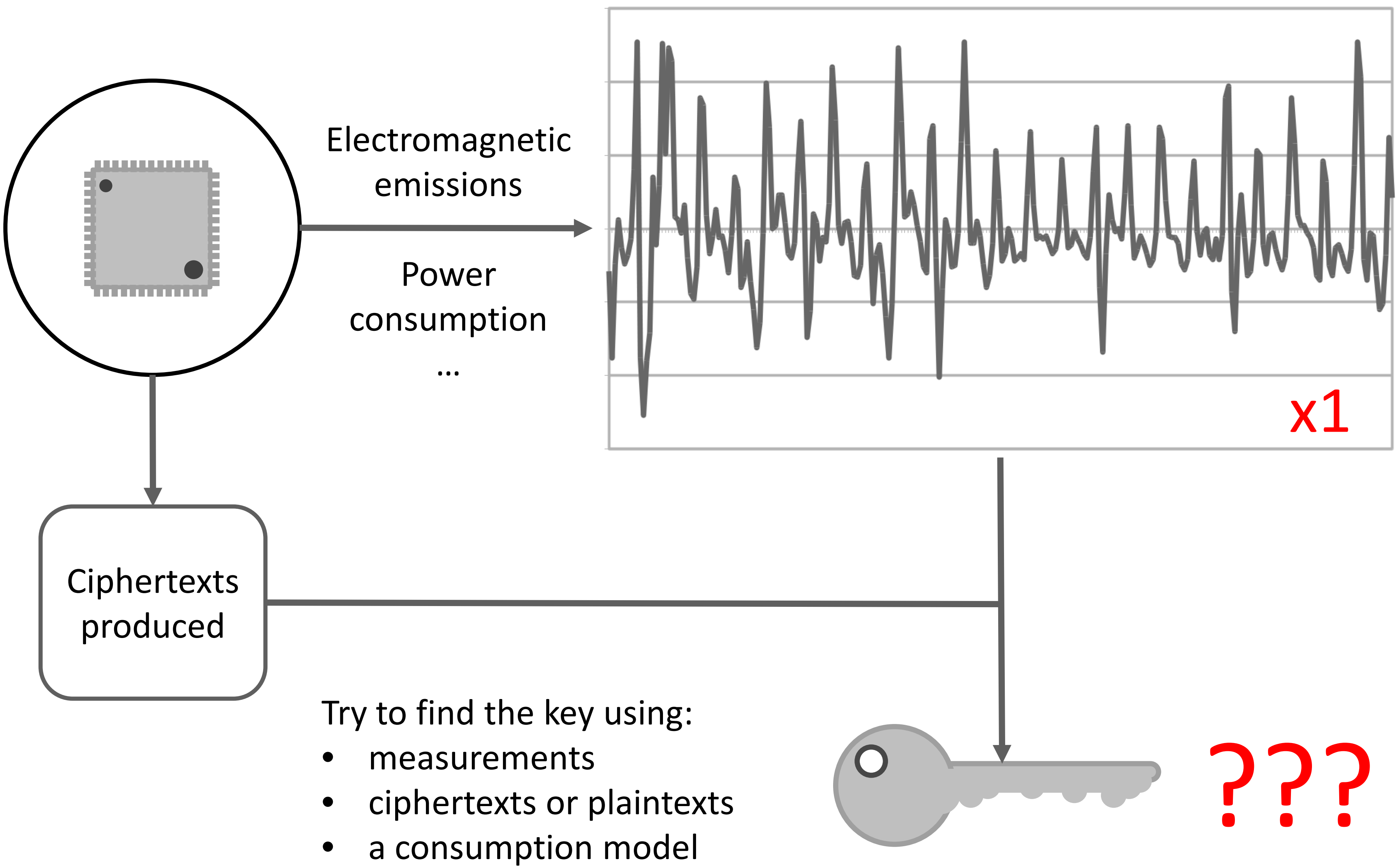
SIDE CHANNEL ATTACKS



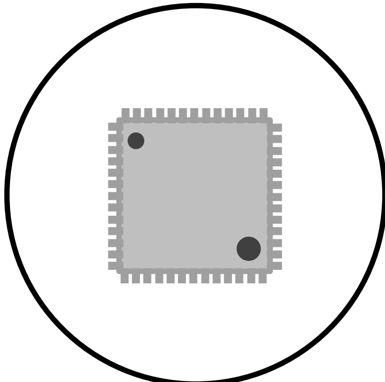
Try to find the key using:

- measurements
- ciphertexts or plaintexts
- a consumption model




Electromagnetic emissions
Power consumption
…
x1
Ciphertexts produced
Try to find the key using:
• measurements
• ciphertexts or plaintexts
• a consumption model
???

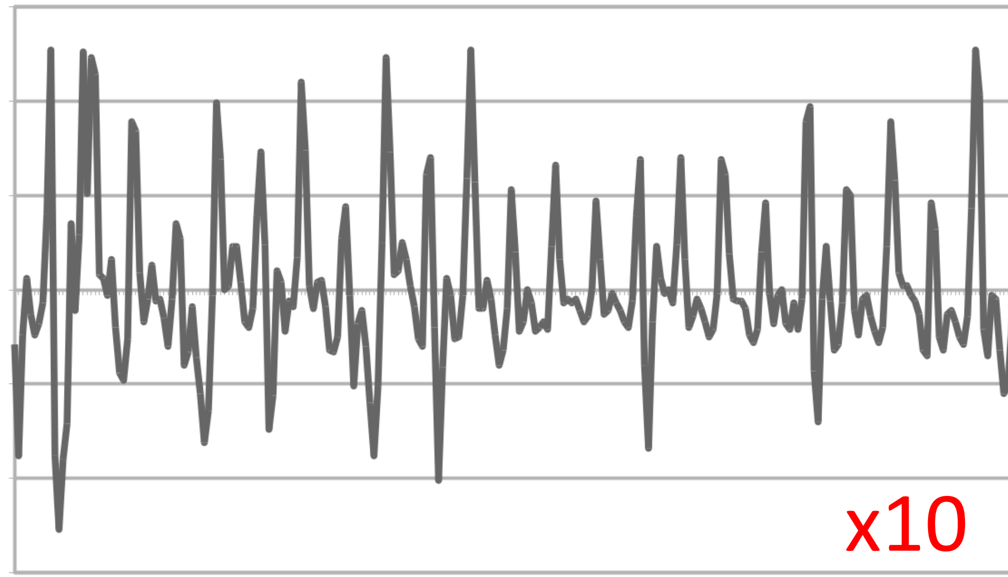
SIDE CHANNEL ATTACKS



Electromagnetic emissions

Power consumption

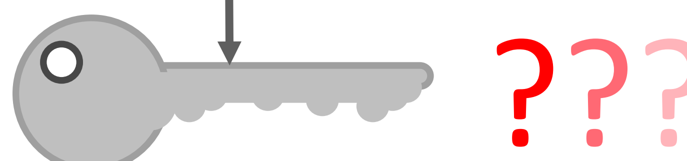
…



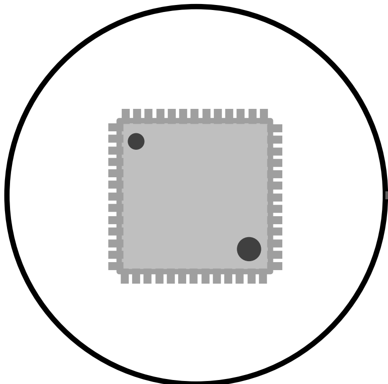
Ciphertexts produced

Try to find the key using:

- measurements
- ciphertexts or plaintexts
- a consumption model



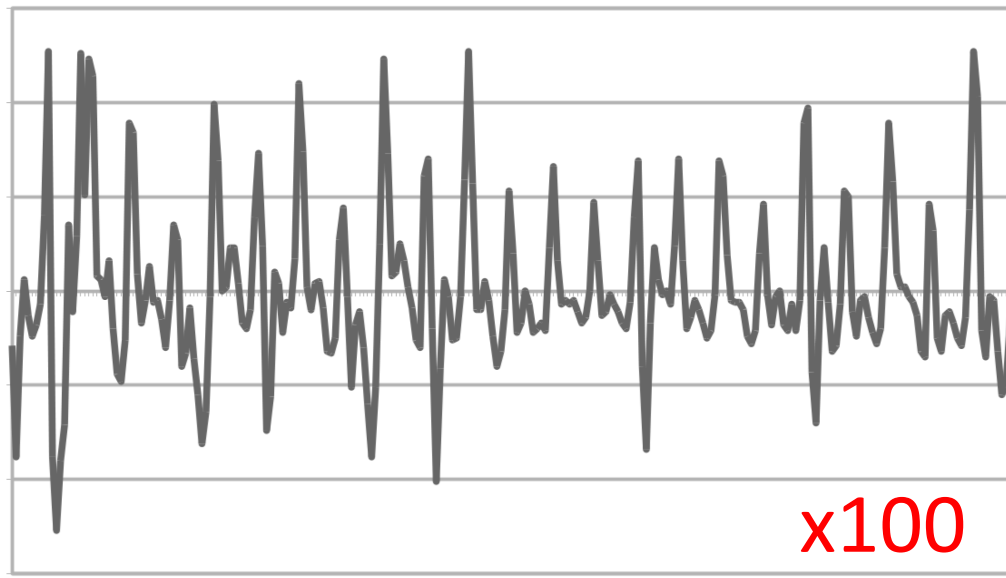
SIDE CHANNEL ATTACKS



Electromagnetic emissions

Power consumption

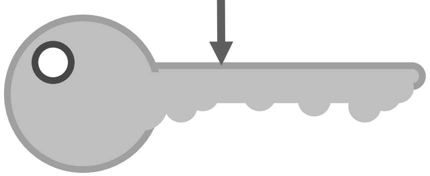
…



Ciphertexts produced

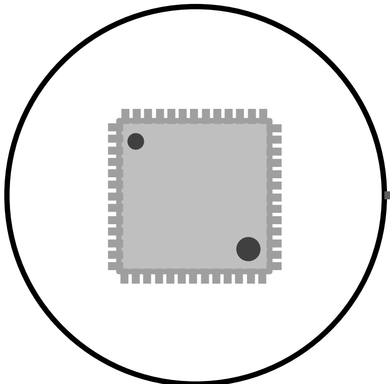
Try to find the key using:

- measurements
- ciphertexts or plaintexts
- a consumption model



??

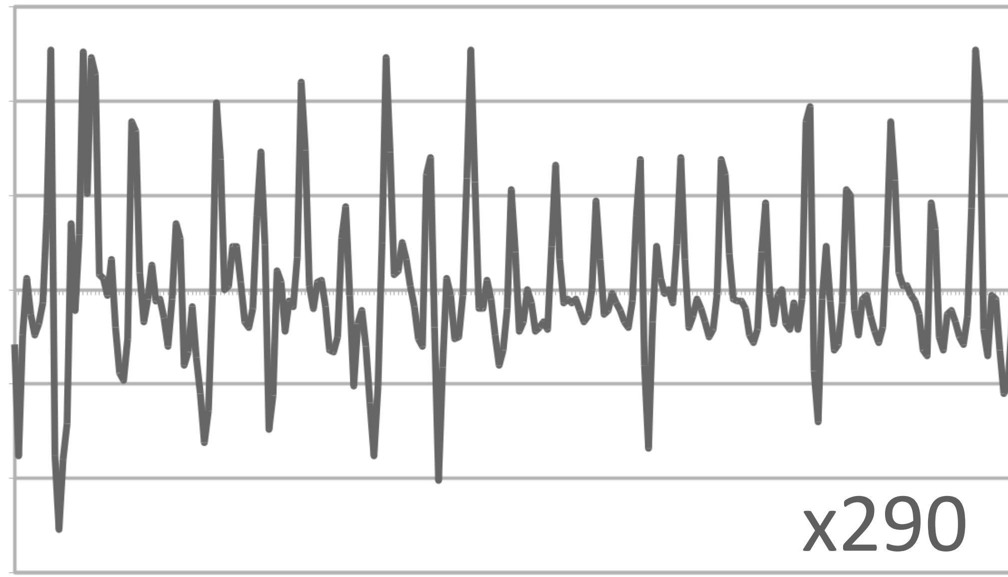
SIDE CHANNEL ATTACKS



Electromagnetic emissions

Power consumption

…

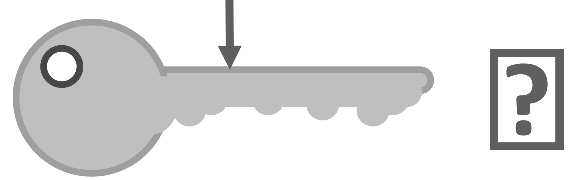


Ciphertexts produced <5kB

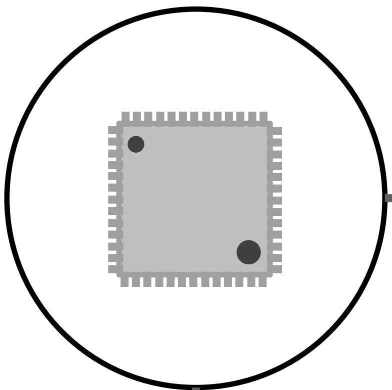
(less than a hello world binary file)

Try to find the key using:

- measurements
- ciphertexts or plaintexts
- a consumption model



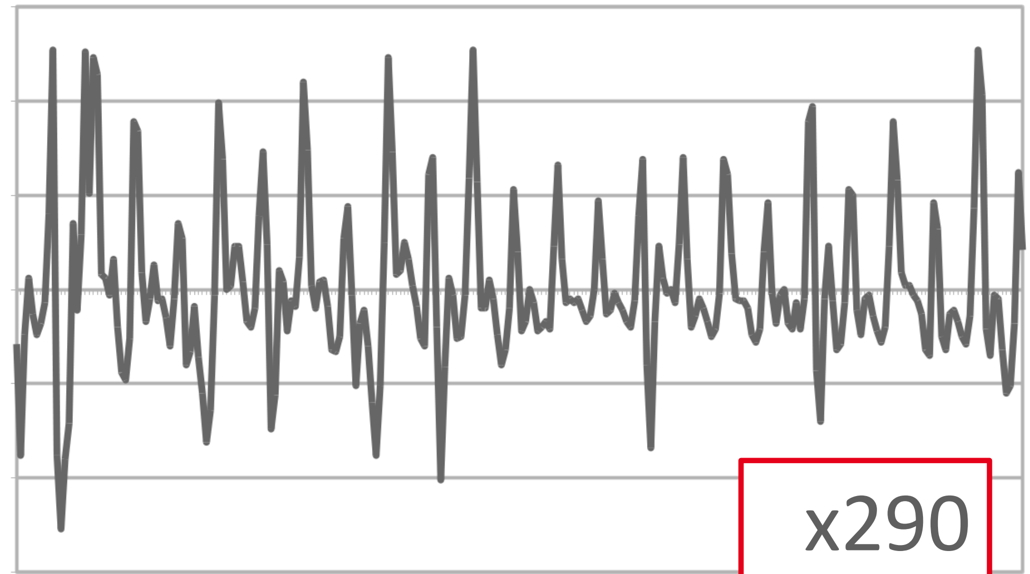
SIDE CHANNEL ATTACKS



Electromagnetic emissions

Power consumption

…



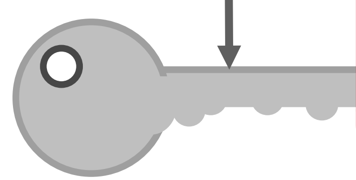
x290

Ciphertexts produced <5kB

(less than a hello world binary file)

Try to find the key using:

- measurements
- ciphertexts or plaintexts
- a consumption model



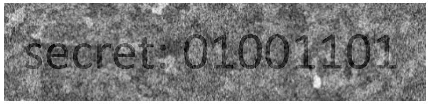
Make this bigger!

SOFTWARE COUNTERMEASURES

secret: 01001101

Hiding

add noise



Measurements are noisy

Masking

split secret into shares

share 1: 10010001
share 2: 11011100

share1 xor share2 = secret

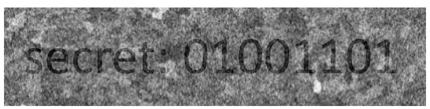
Measurements are no more directly correlated to secret

Attack is harder

secret: 01001101

Hiding

add noise



Masking

split secret
into shares

share 1: 10010001
share 2: 11011100

share1 xor share2 = secret

Measurements
are noisy

Measurements
are no more
directly correlated
to secret

Attack is harder

Electromagnetic emissions
Power consumption

change this

Function's result

don't change this

CODE POLYMORPHISM WITH RUNTIME CODE GENERATION

Electromagnetic emissions
Power consumption

→ depends on →

Executed instructions
Manipulated data

→ impacts →

Function’s result

change this

don’t change this

CODE POLYMORPHISM
WITH RUNTIME CODE GENERATION

Electromagnetic emissions
Power consumption

impacts

Executed instructions
Manipulated data

impacts

Function's result

change this

don't change this

CODE POLYMORPHISM
WITH RUNTIME CODE GENERATION

Electromagnetic emissions
Power consumption

← impacts —

Executed instructions
Manipulated data

— impacts →

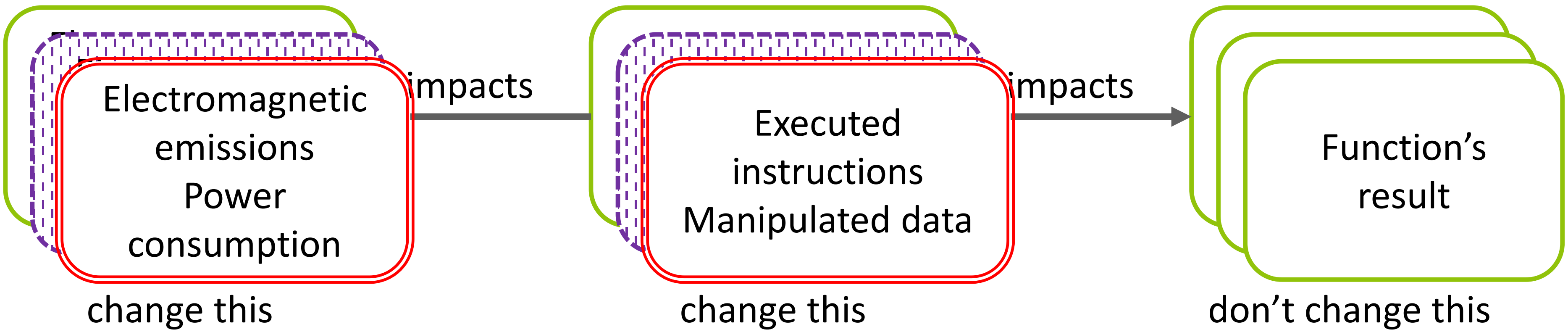
Function's result

change this

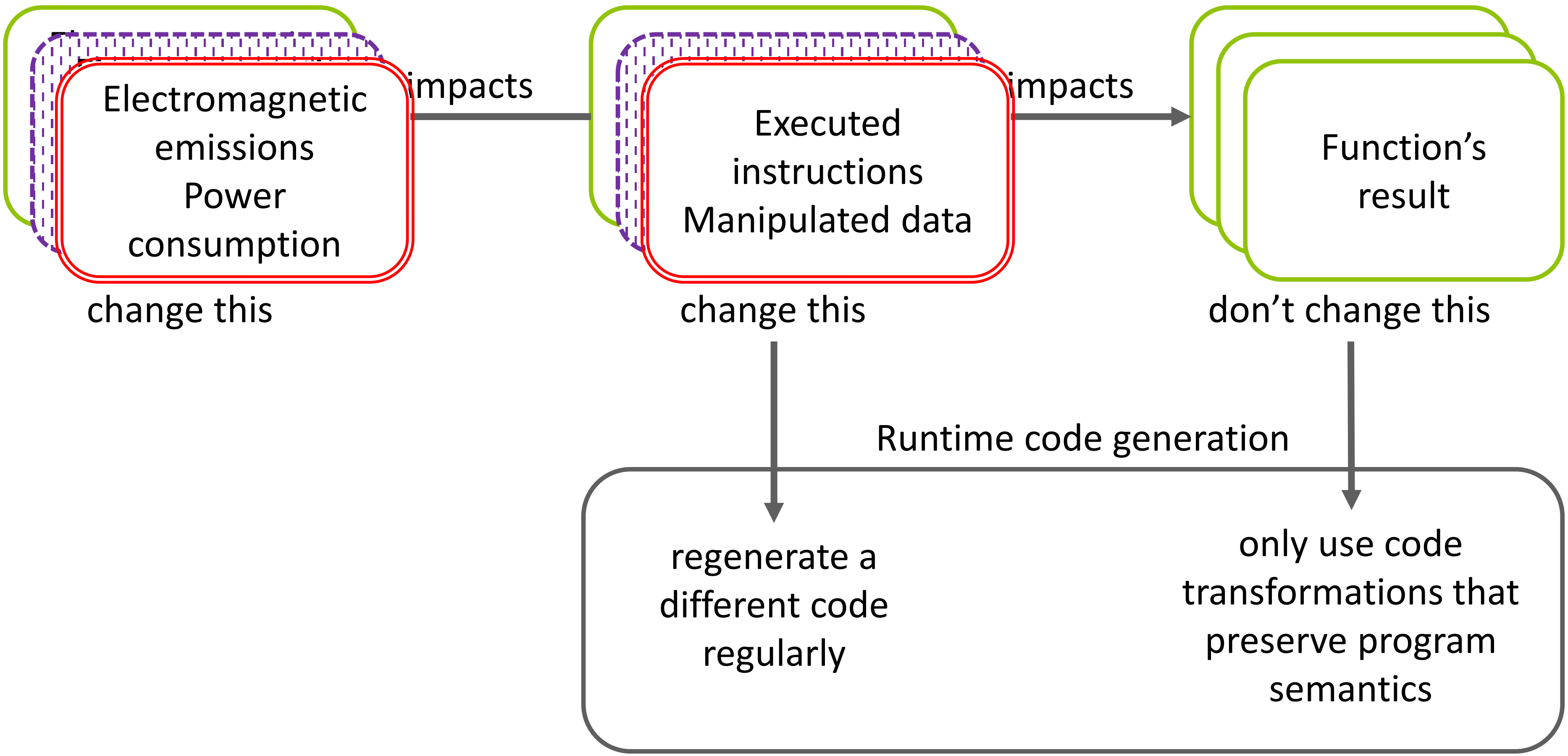
change this

don't change this

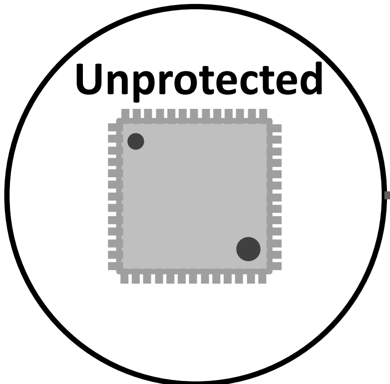
CODE POLYMORPHISM
WITH RUNTIME CODE GENERATION



CODE POLYMORPHISM WITH RUNTIME CODE GENERATION



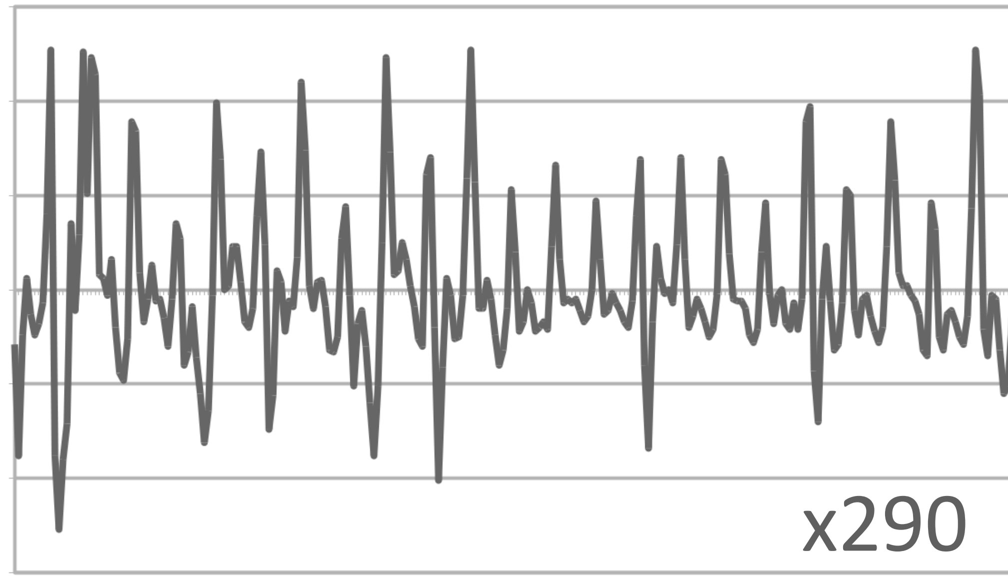
Issues	Our contributions
Countermeasures are usually **manually** applied	**Automatic** application of the countermeasure
Countermeasures are usually given for **particular** ciphers	**Any** code can be hardened
Target a **wide** range of platforms (be lightweight)	Use **static** memory allocation Allocation of a **realistic** size (don’t waste memory) Use **specialized** code generation
An attacker may **write** on an **executable** memory section	Use the **specialization** of the generator to manage memory permissions
Hard to have a **good trade-off** between security and performance	Highly **configurable** → possible to find a trade-off



Electromagnetic emissions

Power consumption

…

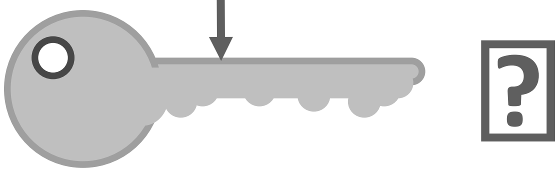


Ciphertexts produced 5kB

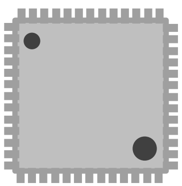
(less than a hello world binary file)

Try to find the key using:

- measurements
- ciphertexts or plaintexts
- a consumption model



Protected

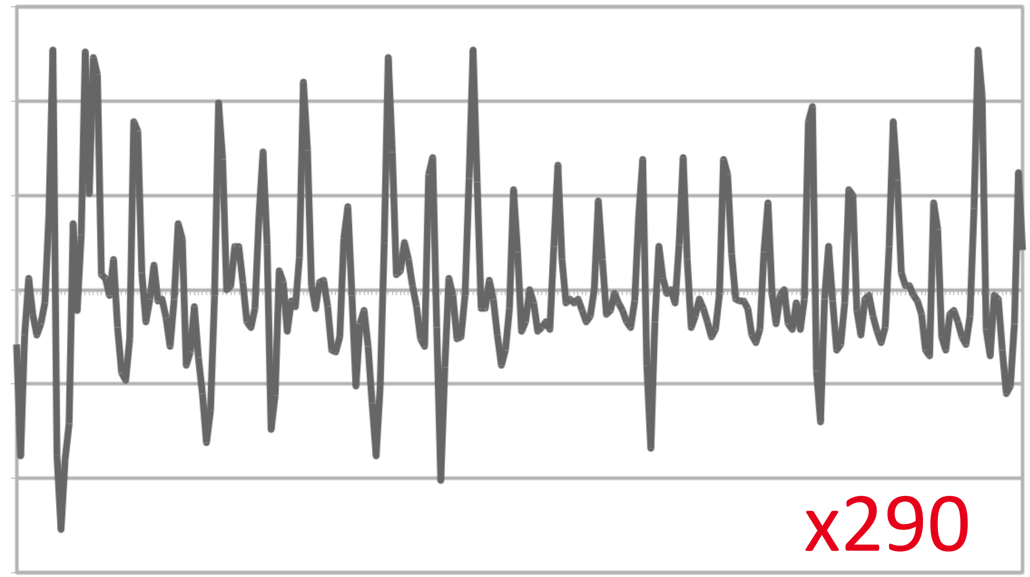


x2.5 overhead

Electromagnetic emissions

Power consumption

…

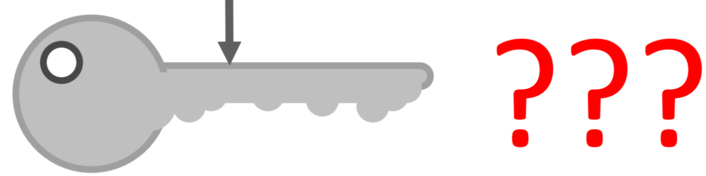


Ciphertexts produced 5kB

(less than a hello world binary file)

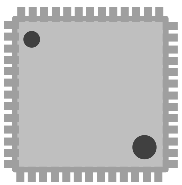
Try to find the key using:

- measurements
- ciphertexts or plaintexts
- a consumption model



SIDE CHANNEL ATTACKS

Protected

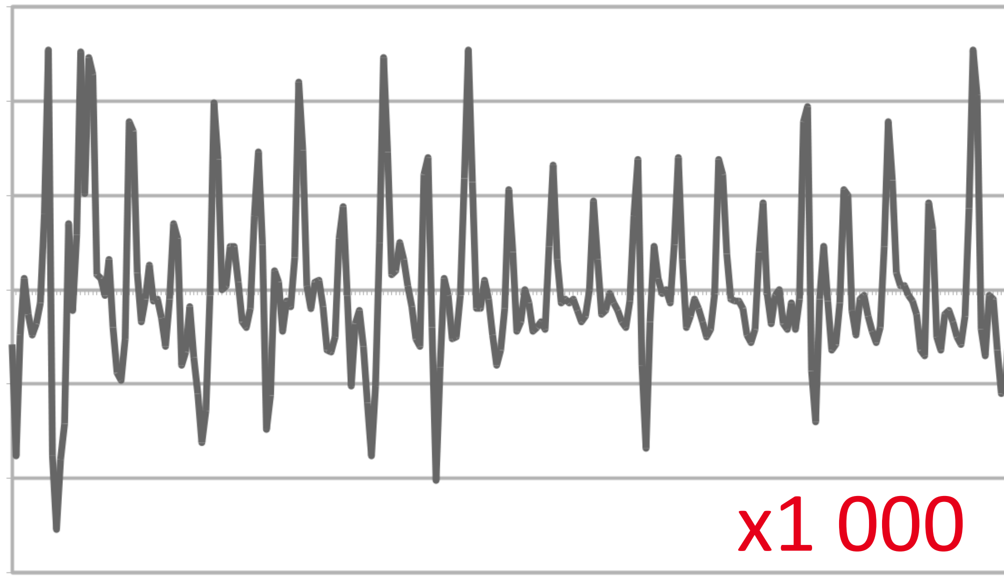


x2.5 overhead

Electromagnetic emissions

Power consumption

…

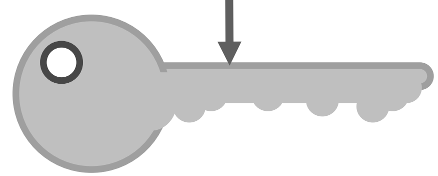


Ciphertexts produced 16kB

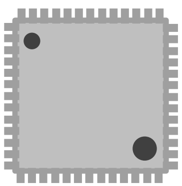
(one second of an mp3 file)

Try to find the key using:

- measurements
- ciphertexts or plaintexts
- a consumption model



Protected

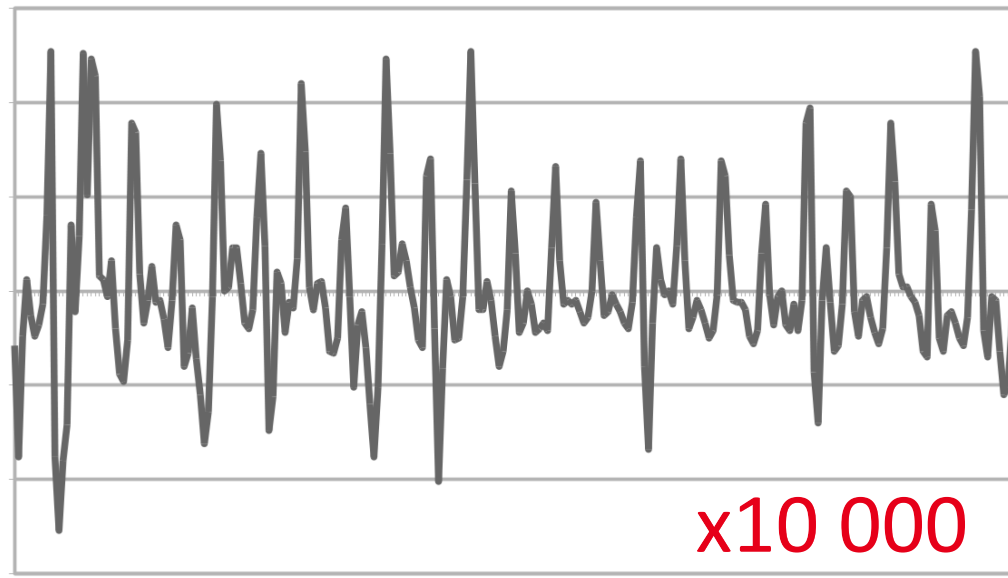


x2.5 overhead

Electromagnetic emissions

Power consumption

…

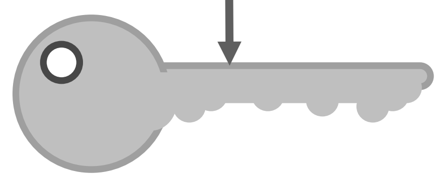


Ciphertexts produced 160kB

(10 seconds of an mp3 file)

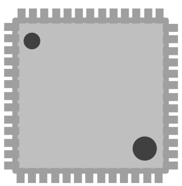
Try to find the key using:

- measurements
- ciphertexts or plaintexts
- a consumption model



SIDE CHANNEL ATTACKS

Protected

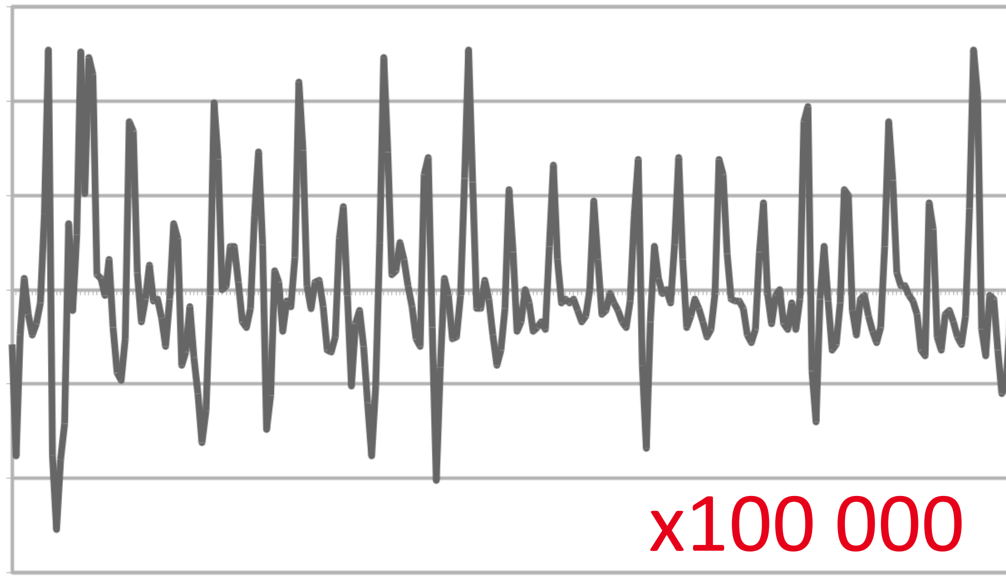


x2.5 overhead

Electromagnetic emissions

Power consumption

…

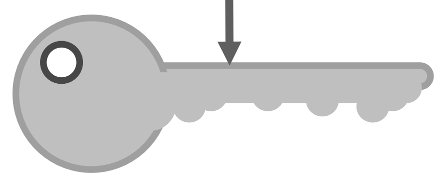


Ciphertexts produced 1.6MB

(our paper)

Try to find the key using:

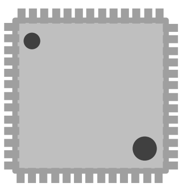
- measurements
- ciphertexts or plaintexts
- a consumption model



???

SIDE CHANNEL ATTACKS

Protected

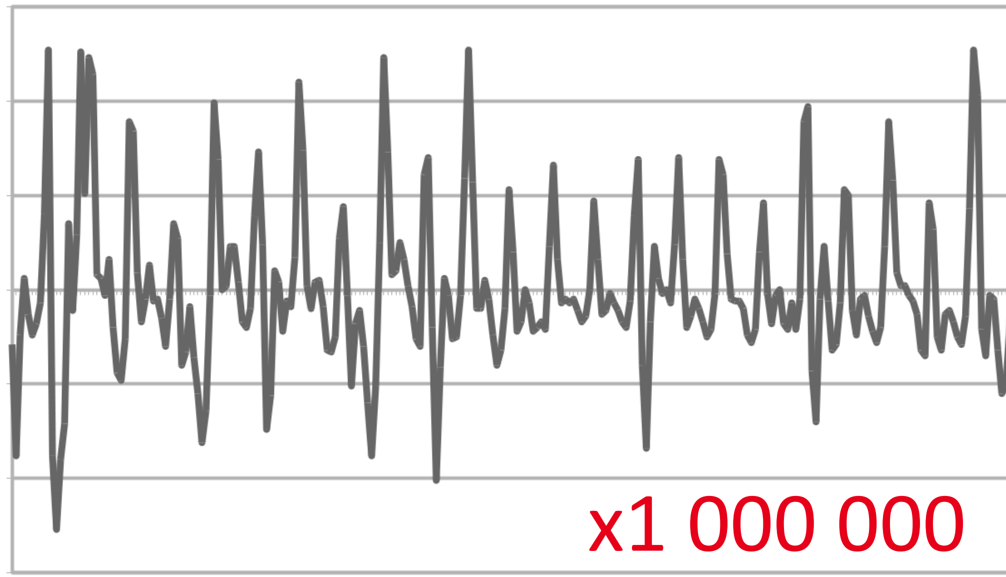


x2.5 overhead

Electromagnetic emissions

Power consumption

…

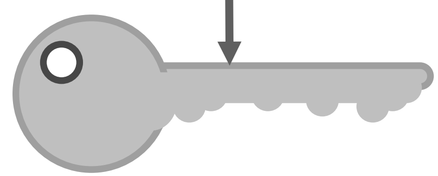


Ciphertexts produced 16MB

(a 20Mpixels jpeg image)

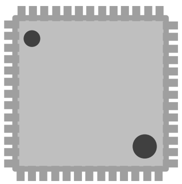
Try to find the key using:

- measurements
- ciphertexts or plaintexts
- a consumption model



SIDE CHANNEL ATTACKS

Protected

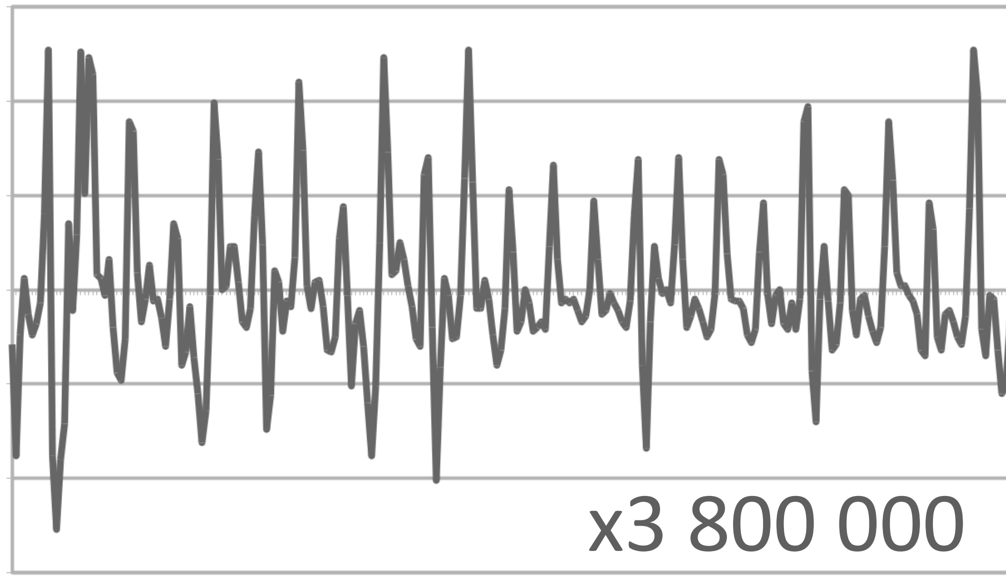


x2.5 overhead

Electromagnetic emissions

Power consumption

…

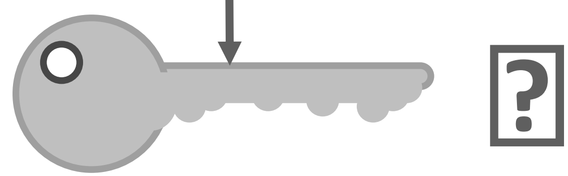


Ciphertexts produced 60MB

(a 3minutes long full HD video)

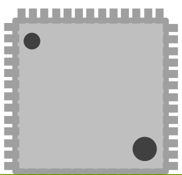
Try to find the key using:

- measurements
- ciphertexts or plaintexts
- a consumption model



SIDE CHANNEL ATTACKS

Protected

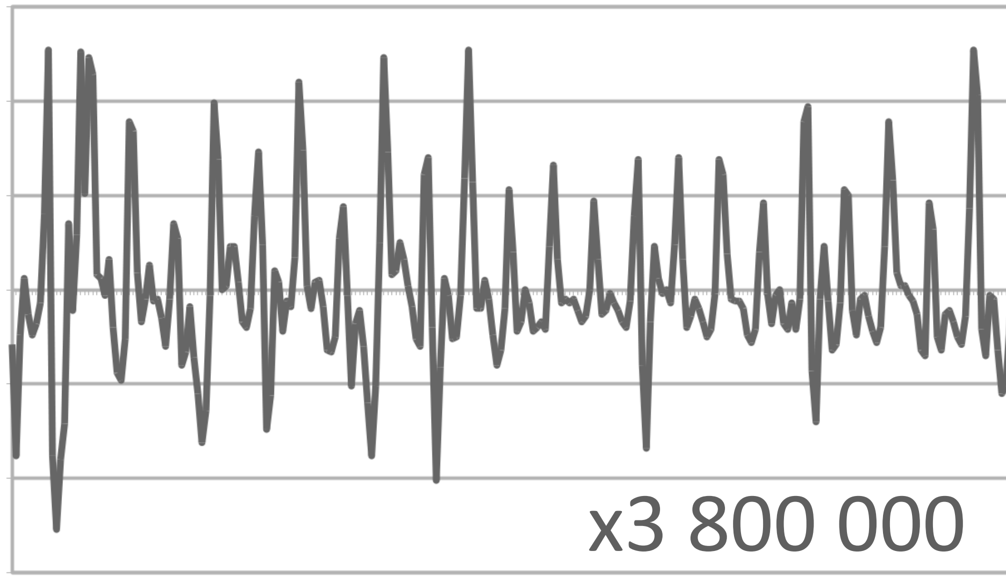


x2.5 overhead

Electromagnetic emissions

Power consumption

…



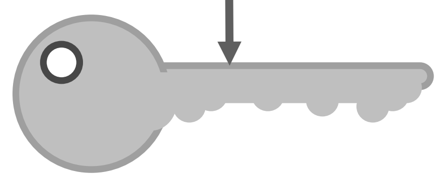
Ciphertexts produced 60MB

(a 3minutes long full HD video)

x13 000 improvement!

Try to find the key using:

- measurements
- ciphertexts or plaintexts
- a consumption model

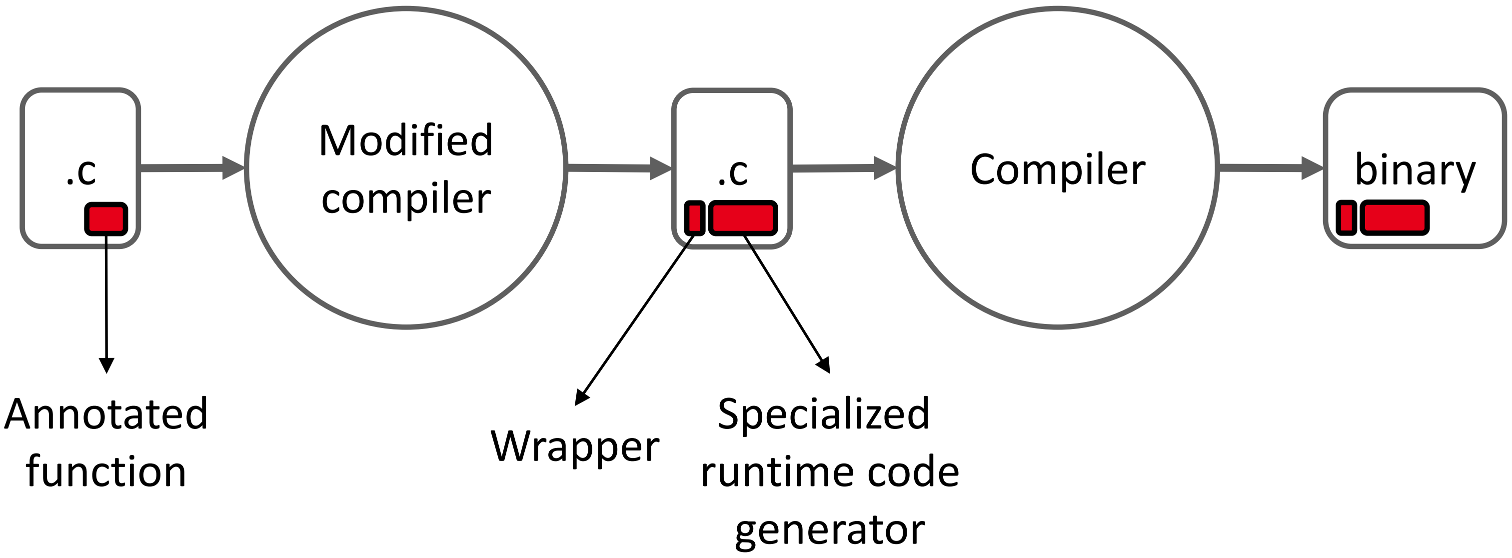


?

- **Background**
  - Side channel attacks
  - Software countermeasures
  - Code polymorphism
- **Automated application of code polymorphism**
  - Overview
  - Code transformations used
  - Memory management
- **Experimental evaluation**
  - Performance evaluation
  - Security evaluation

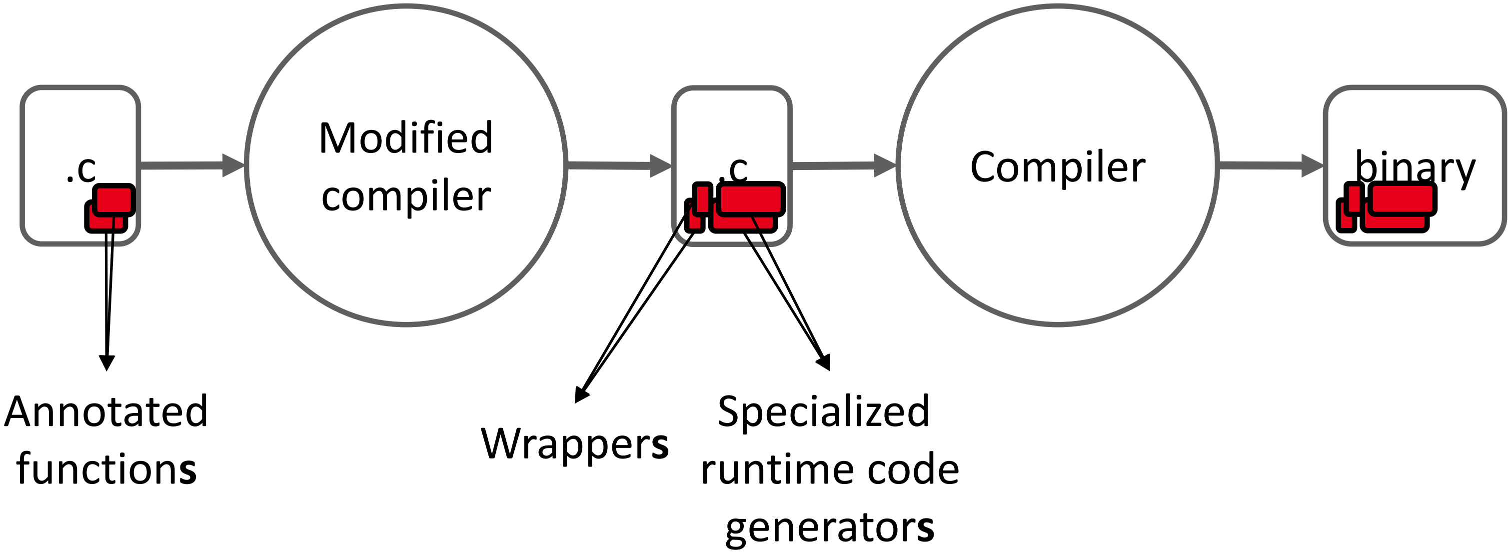
- **Background**
  - Side channel attacks
  - Software countermeasures
  - Code polymorphism
- **Automated application of code polymorphism**
  - Overview
  - Code transformations used
  - Memory management
- **Experimental evaluation**
  - Performance evaluation
  - Security evaluation

STATICALLY



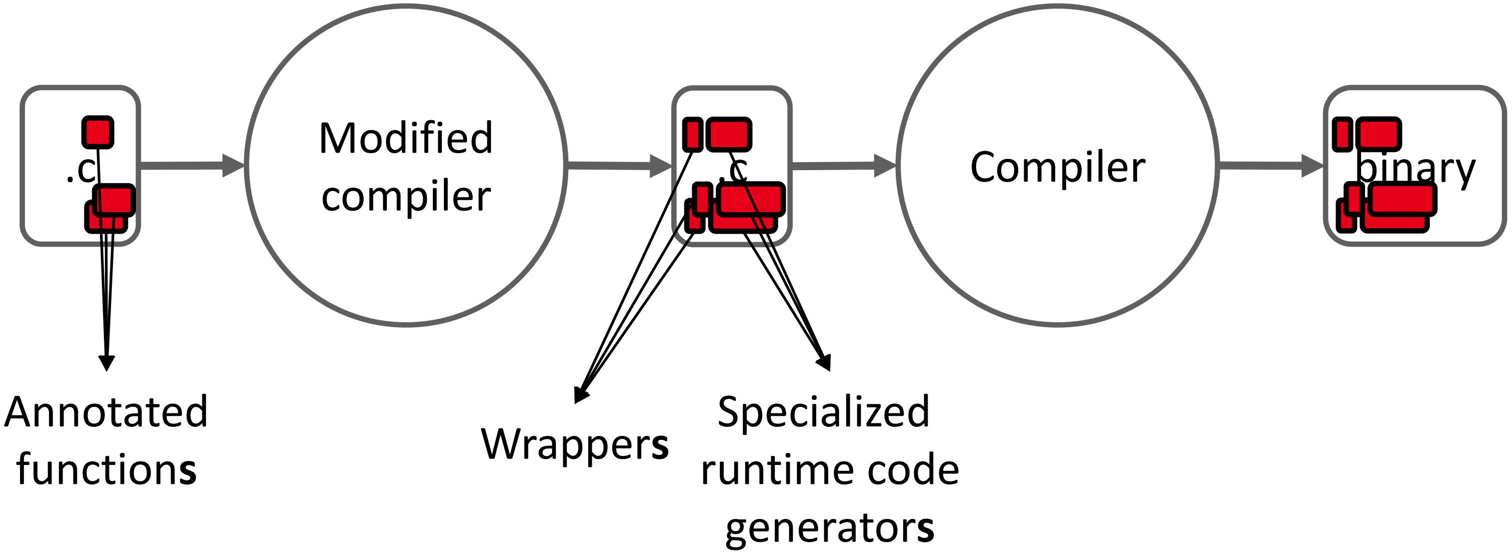
Main idea:
The annotated function is replaced by a wrapper and a generator

STATICALLY



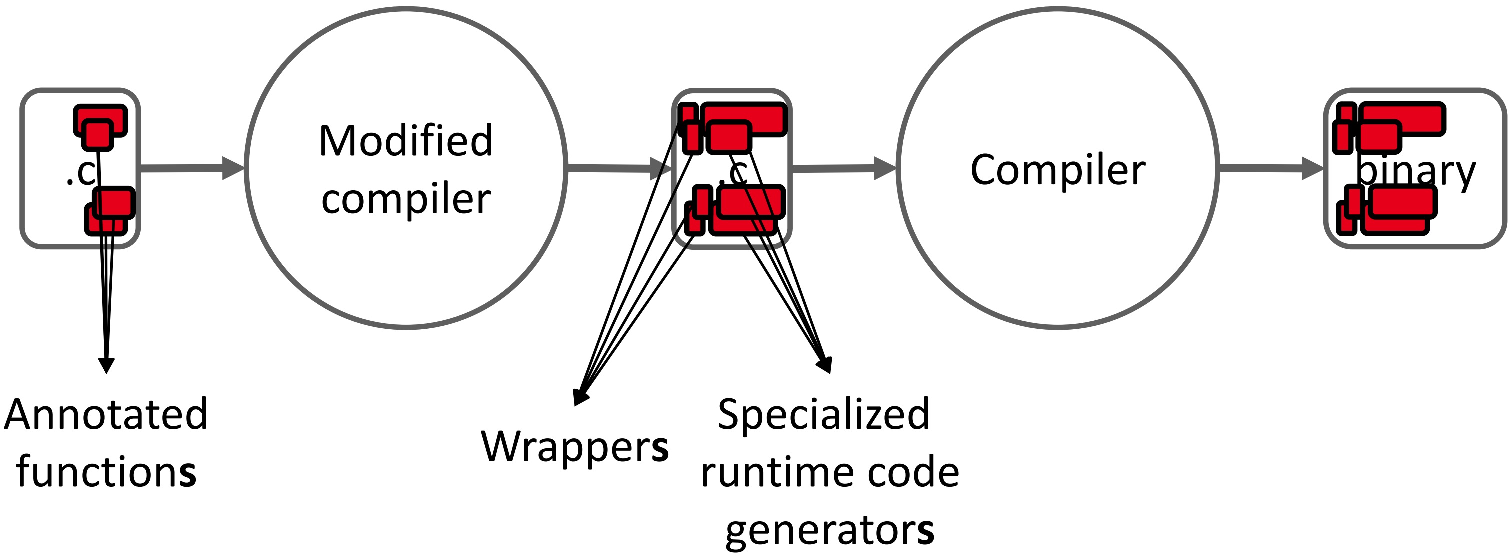
Main idea:
Each annotated function has its own generator (with shared code segments)

STATICALLY



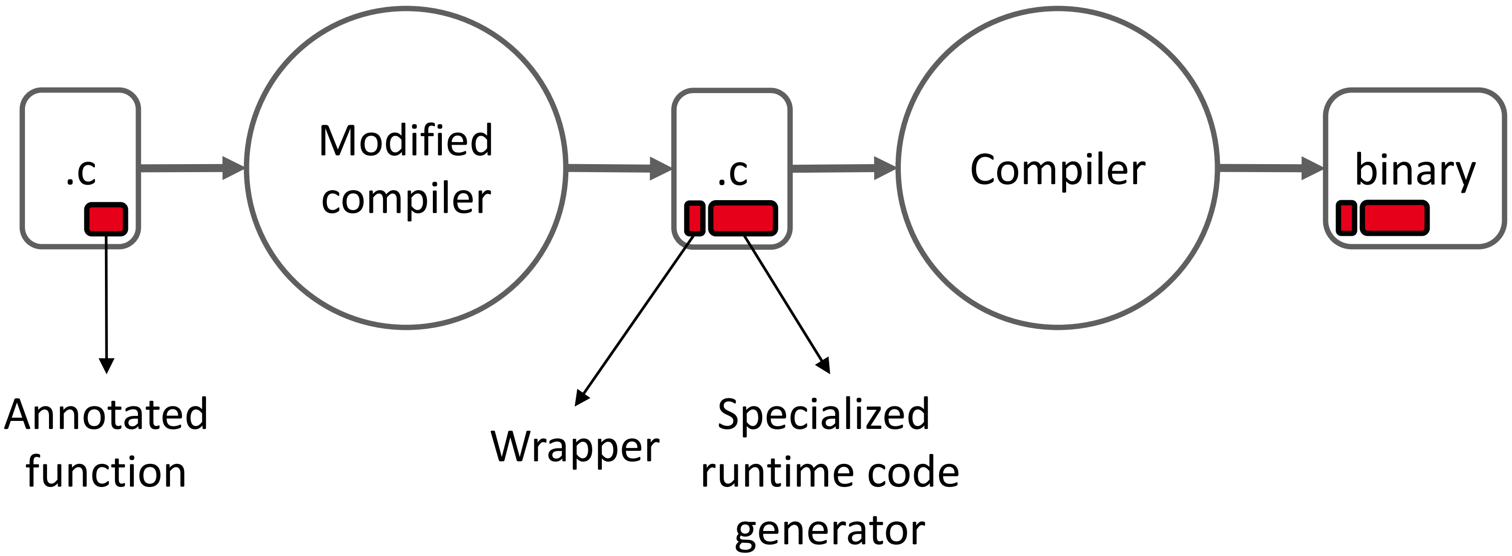
Main idea:
Each annotated function has its own generator
(with shared code segments)

STATICALLY

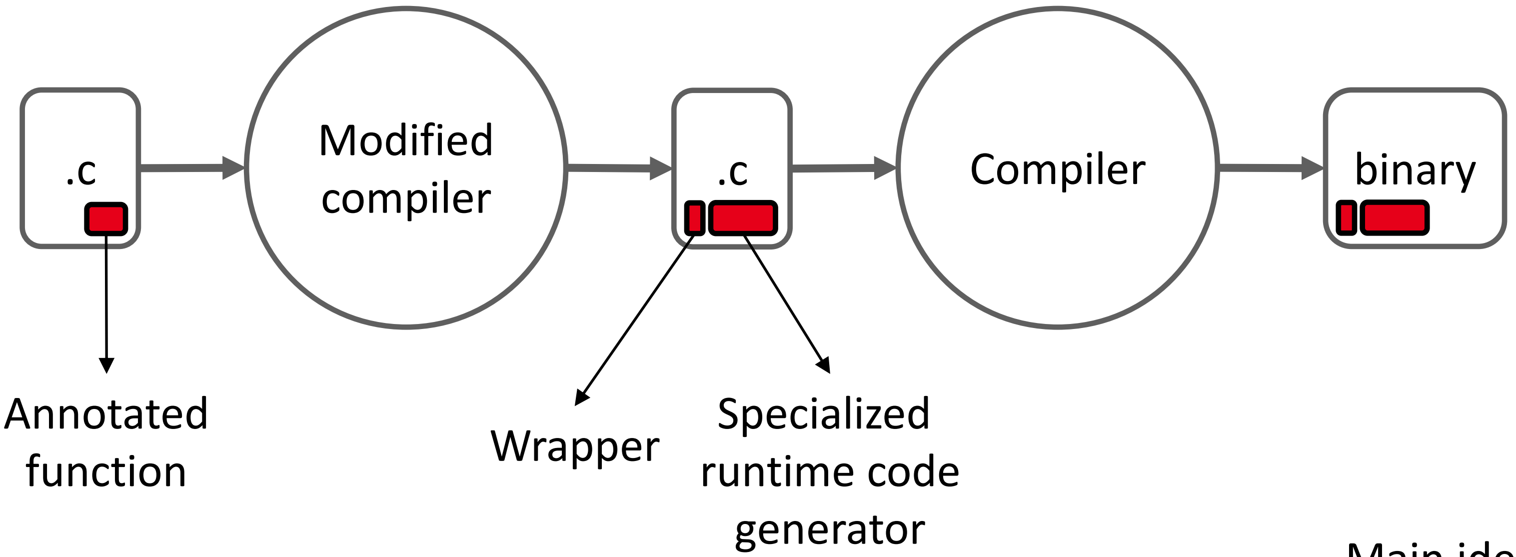


Main idea:
Each annotated function has its own generator (with shared code segments)

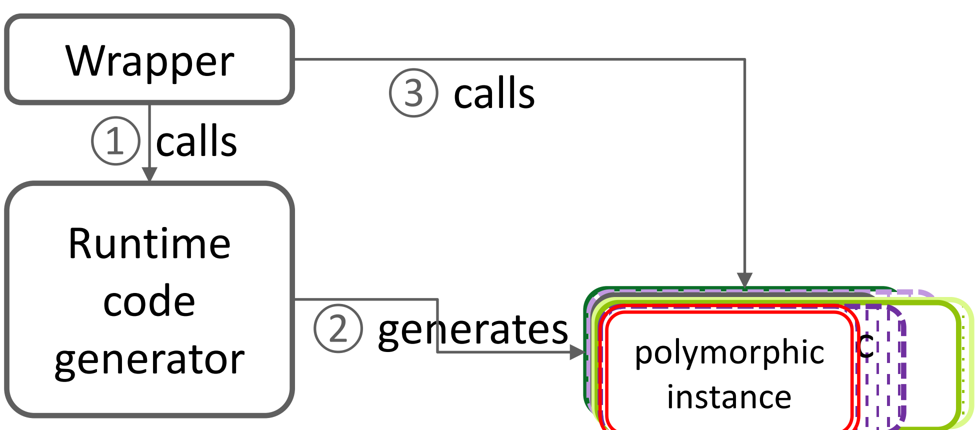
STATICALLY


.c
Modified compiler
.c
Compiler
binary
Annotated function
Wrapper
Specialized runtime code generator

STATICALLY

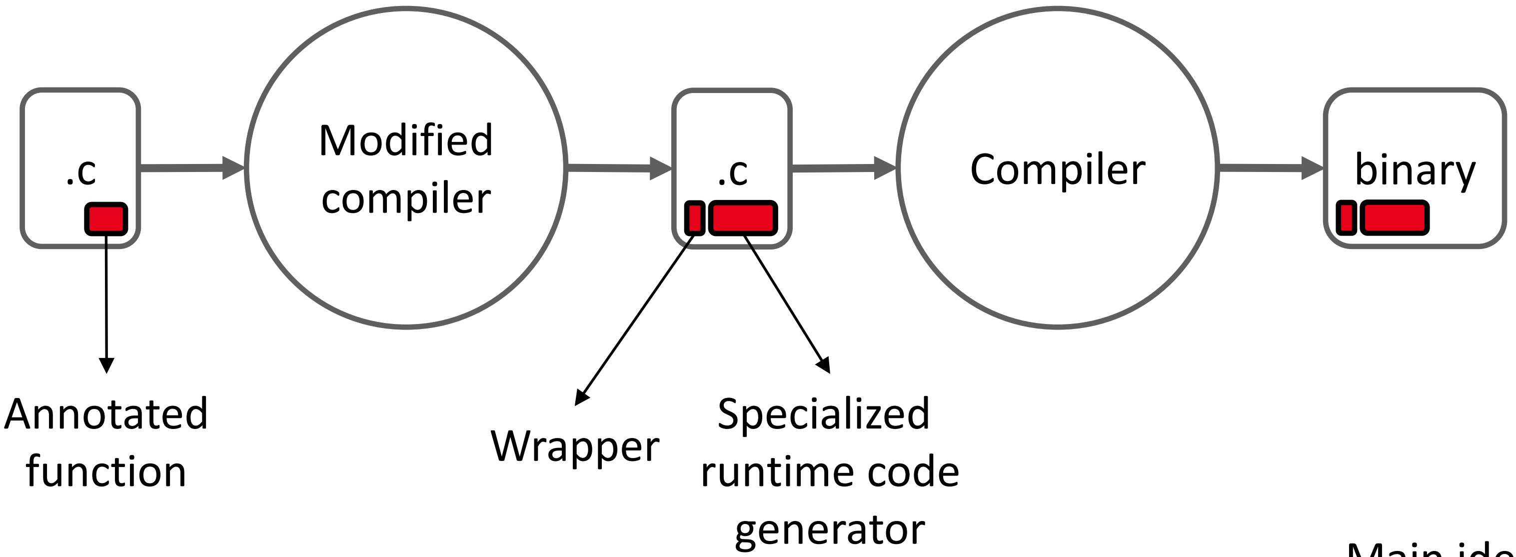


RUNTIME

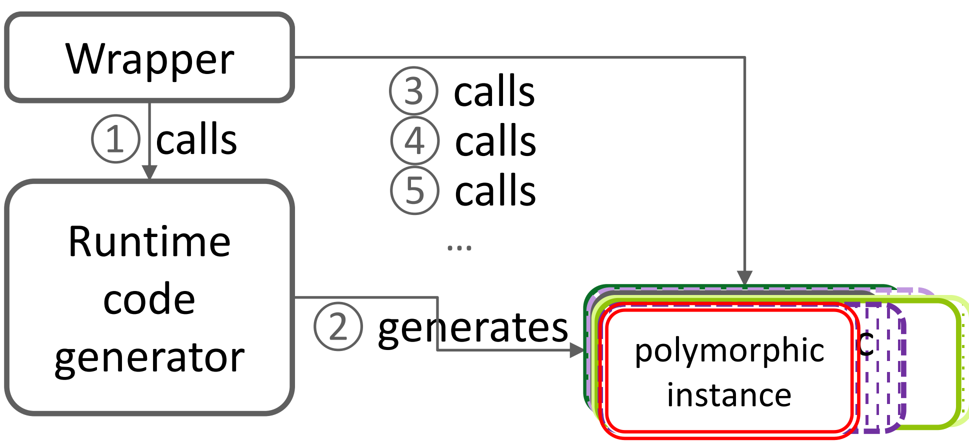


Main idea:
At runtime, a new polymorphic instance is generated at each call

STATICALLY



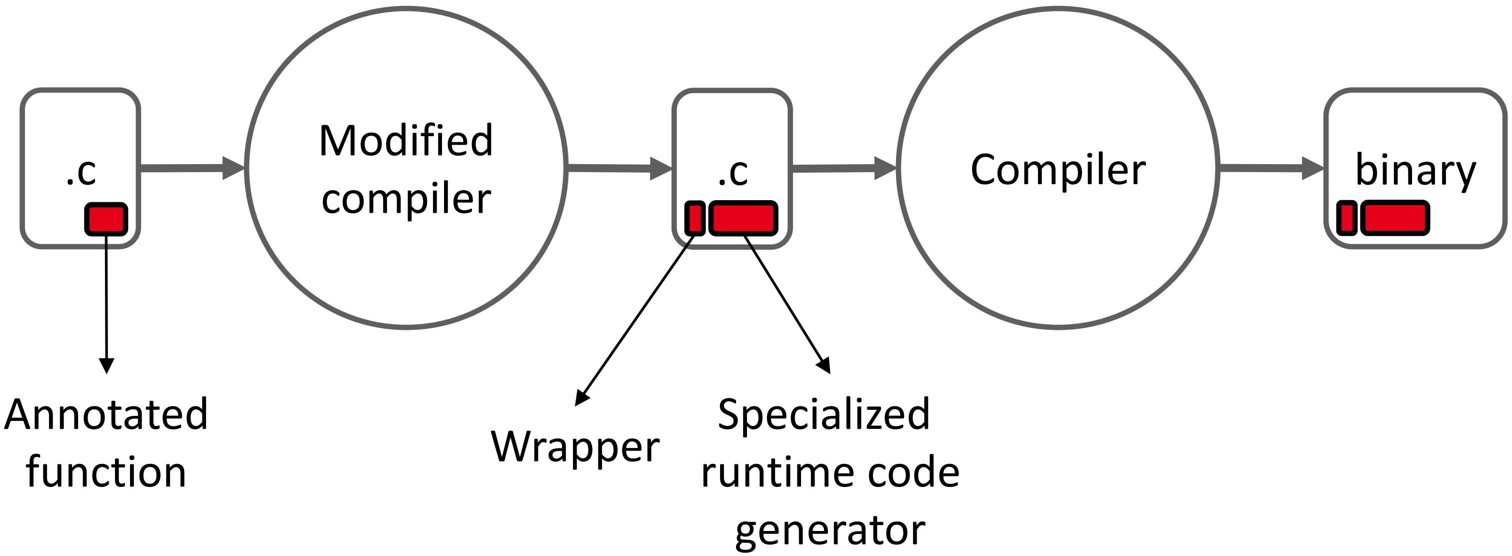
RUNTIME



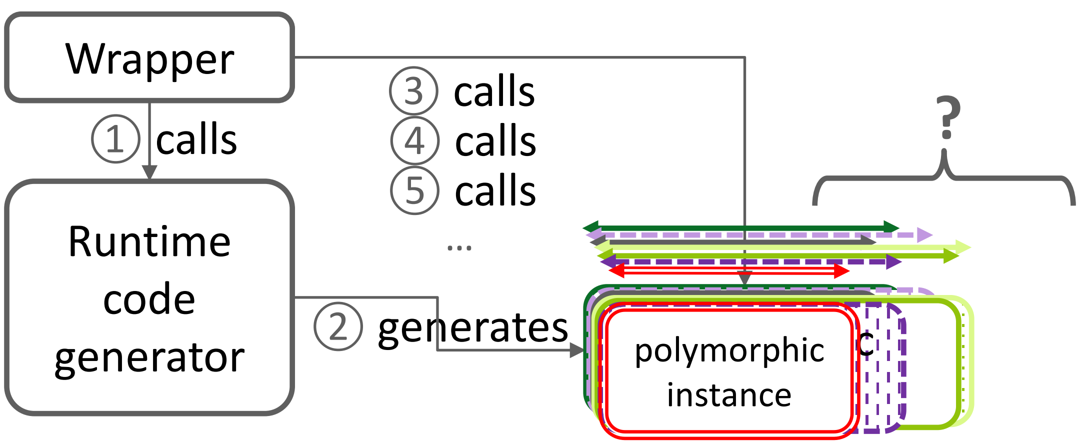
Main idea:
At runtime, a new polymorphic instance is generated ~~at each call~~ once in a while

How to find a good trade-off between security and performance?
How to have variability in between generations?

STATICALLY



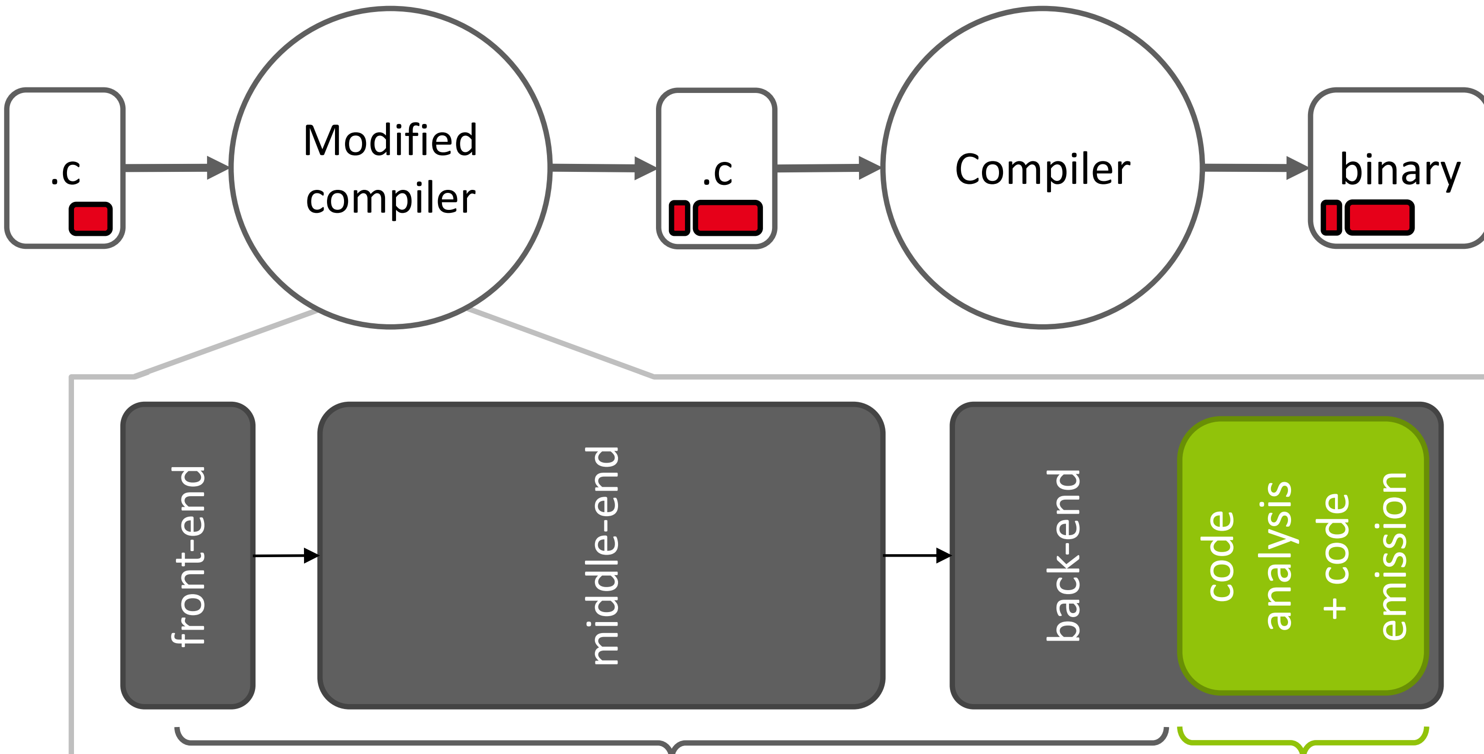
RUNTIME



Main idea:
The size of polymorphic instances vary

How to allocate memory?

STATICALLY

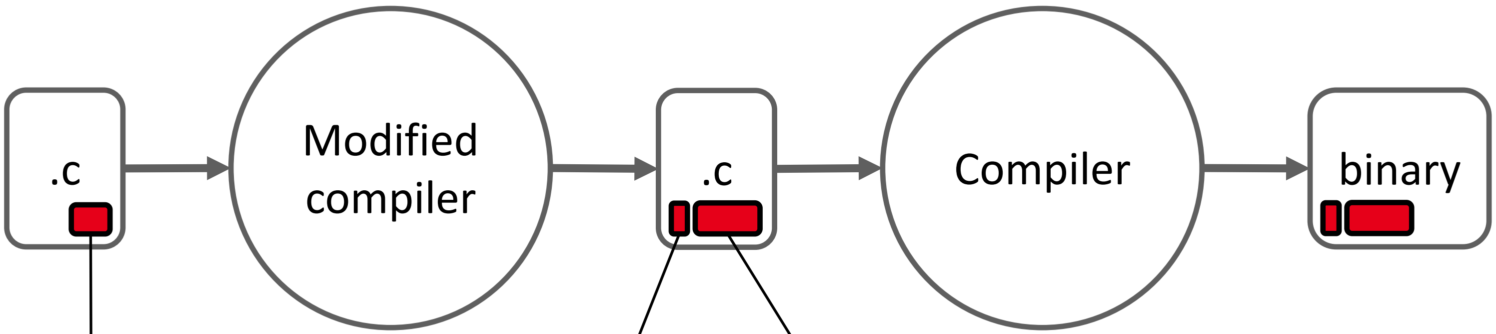


All standard compiler optimizations are applied
→ the runtime generated code is statically optimized

We emit C code instead of ARM Thumb assembly

We perform static analysis to help the runtime code generator (memory and register)

STATICALLY




```
#pragma odo_polymorphic
int f_critical(int a, int b) {
        int c = a^b;
        a = a+b;
}
```

```
int f_critical(int a, int b) {
        if (SHOULD_BE_REGENERATED())
                SGPC_f_critical();
        return code_f(a, b);
}
```

```
void SGPC_f_critical() {
        raise_interrupt_rm_X_add_W(code_f);
        reg_t r[] = {0,1,2,3,4,5,6,...,12,13,14,15};
        push_T2_callee_saved_registers();
        eor_T2(r[4], r[1], r[0]);
        add_T2(r[0], r[1], r[0]);
        sdiv_T2(r[1], r[0], r[4]);
        mls_T2(r[0], r[1], r[4], r[0]);
        pop_T2_callee_saved_registers();
        raise_interrupt_rm_W_add_X(code_f);
}
```

- **Background**
  - Side channel attacks
  - Software countermeasures
  - Code polymorphism

- **Automated application of code polymorphism**
  - Overview
  - Code transformations used
  - Memory management

- **Experimental evaluation**
  - Performance evaluation
  - Security evaluation

CODE TRANSFORMATIONS USED AT RUNTIME

add r4, r4, r5 xor r6, r5, r8	**Register shuffling** RANDOM general purpose register permutation r4 → r11 ... add r11, r11, r7 xor r8, r7, r5	**Instruction shuffling** independent instructions are emitted in a RANDOM order xor r6, r5, r8 add r4, r4, r5

Semantic variants
replacement of an instruction by a RANDOMLY selected semantic variant
add r4, r4, r5
xor r6, r5, #12348
xor r6, r6, r8
xor r6, r6, #12348
	Noise instructions
	insertion of a RANDOM number of RANDOMLY chosen noise instructions
	add r4, r4, r5
	sub r7, r6, r2
	load r3, r10, #53
	xor r6, r5, r8
	useless instructions
		Dynamic noise
		RANDOM insertion of noise instructions with a RANDOM jump
		add r4, r4, r5
		jump 0, 1 or 2 instructions
		sub r7, r6, r2
		load r3, r10, #53
		xor r6, r5, r8

Period of regeneration $\mathbb{N}$	**Register shuffling** $\{0, 1\}$	**Instruction shuffling** $\{0, 1\}$
Semantic variants $\{0, 1, 2\}$	**Noise instructions** $\{0, 1, 2\} \times \mathbb{R} \times \mathbb{N}$	**Dynamic noise** $\mathbb{N}$

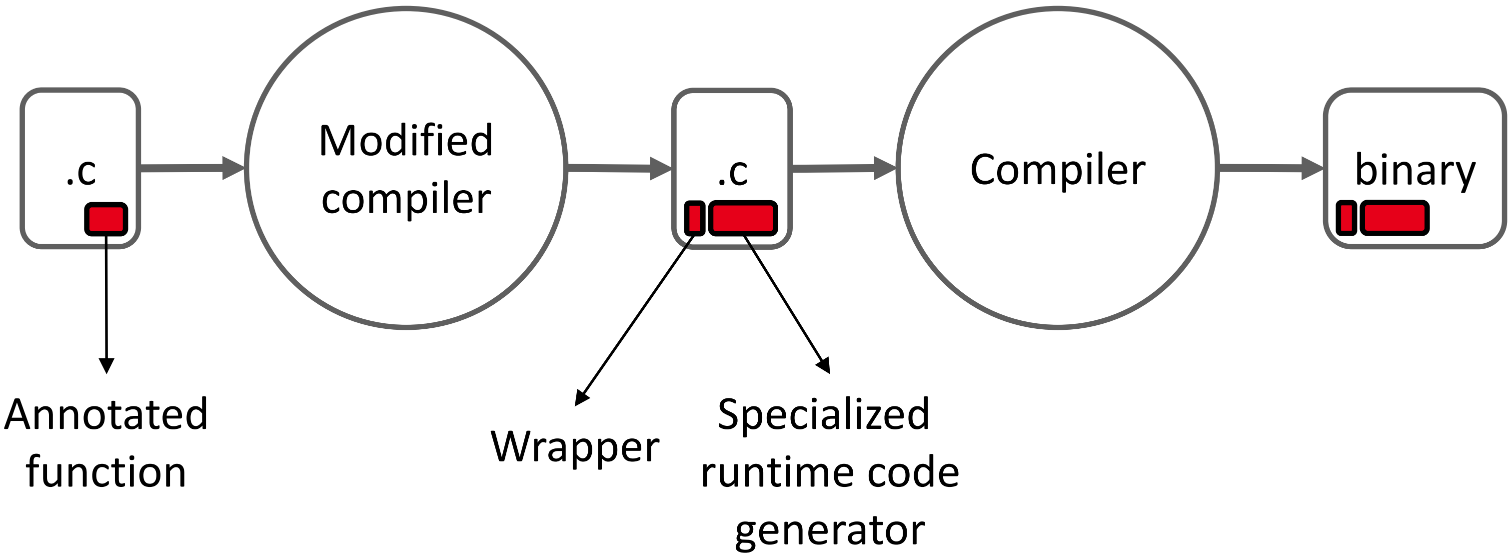
Total configuration space:

$$\{0, 1\}^2 \times \{0, 1, 2\}^2 \times \mathbb{R} \times \mathbb{N}^3$$

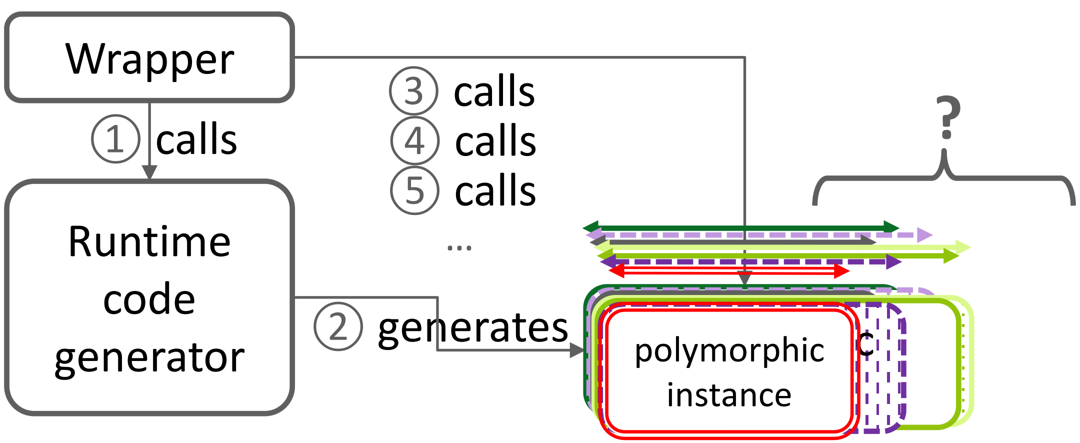
OUTLINE

- **Background**
  - Side channel attacks
  - Software countermeasures
  - Code polymorphism
- **Automated application of code polymorphism**
  - Overview
  - Code transformations used
  - Memory management
    - Memory allocation & overflow prevention
    - Memory permissions
- **Experimental evaluation**
  - Performance evaluation
  - Security evaluation

STATICALLY



RUNTIME

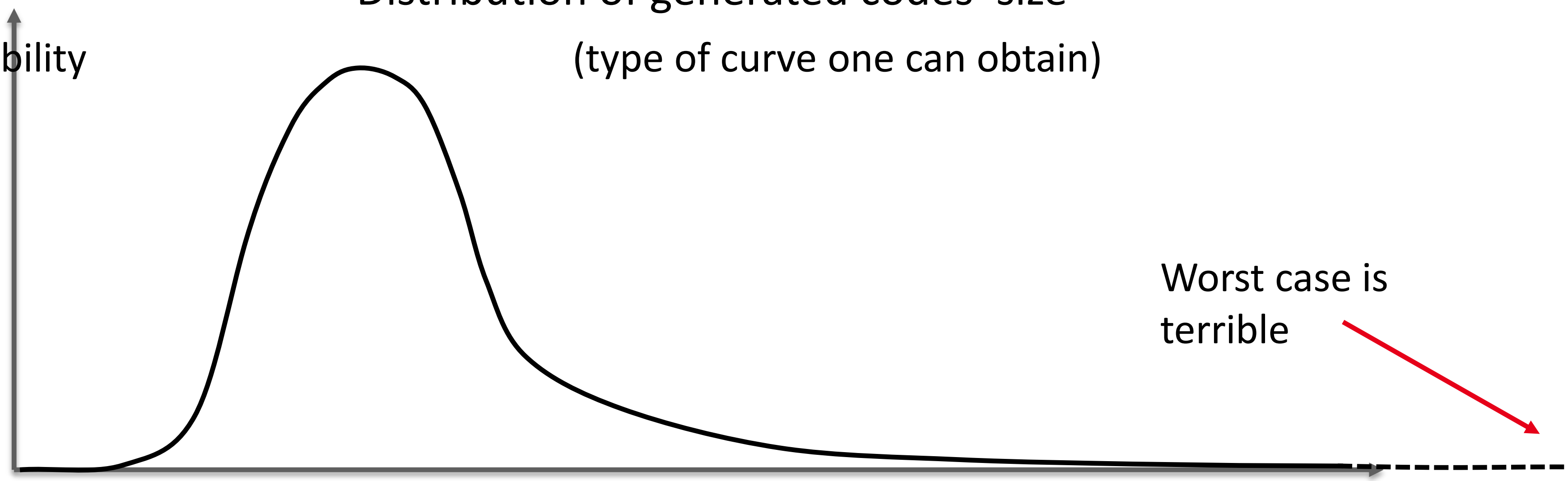


Main idea:
The size of polymorphic instances vary

How to allocate memory?

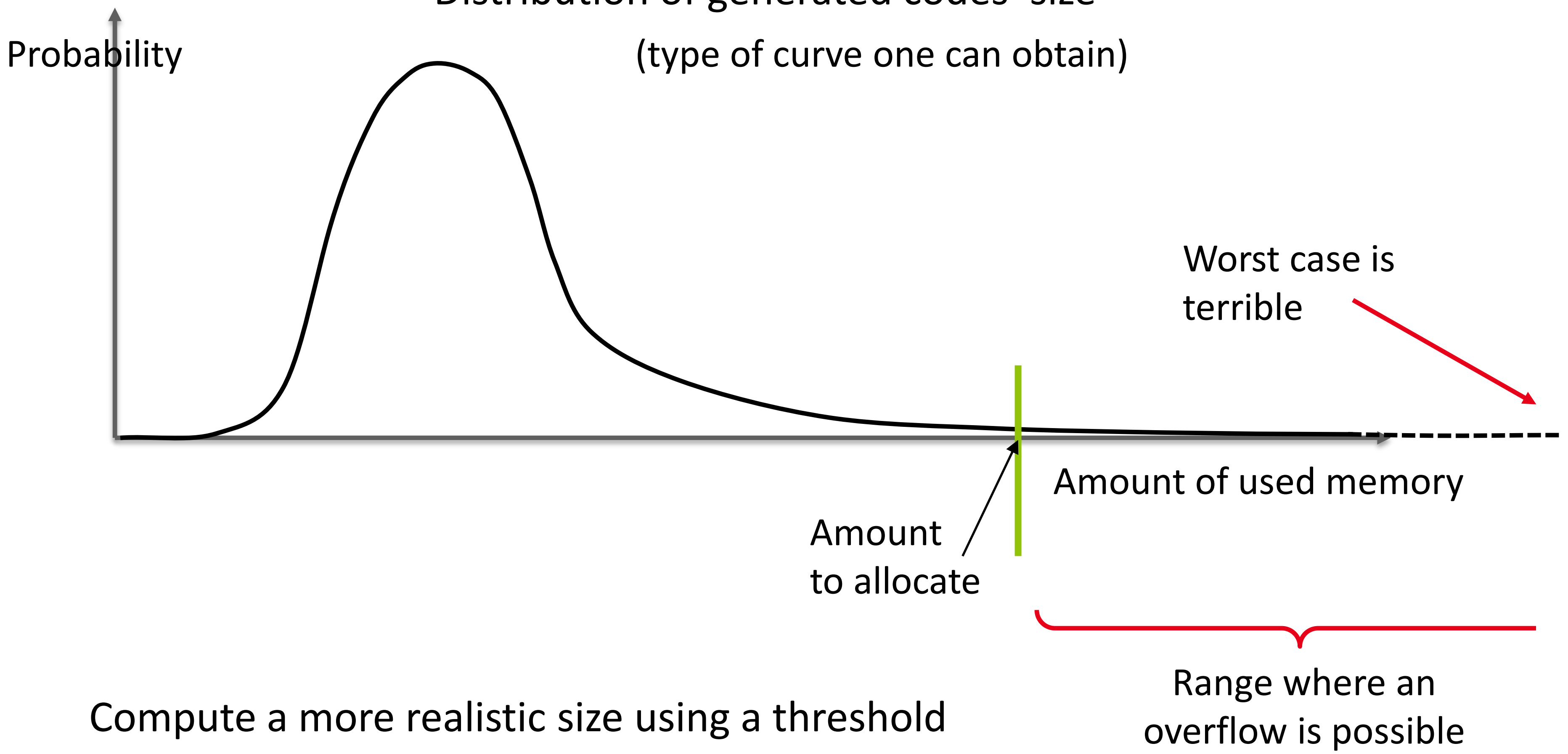
Distribution of generated codes' size

(type of curve one can obtain)



Distribution of generated codes' size

(type of curve one can obtain)



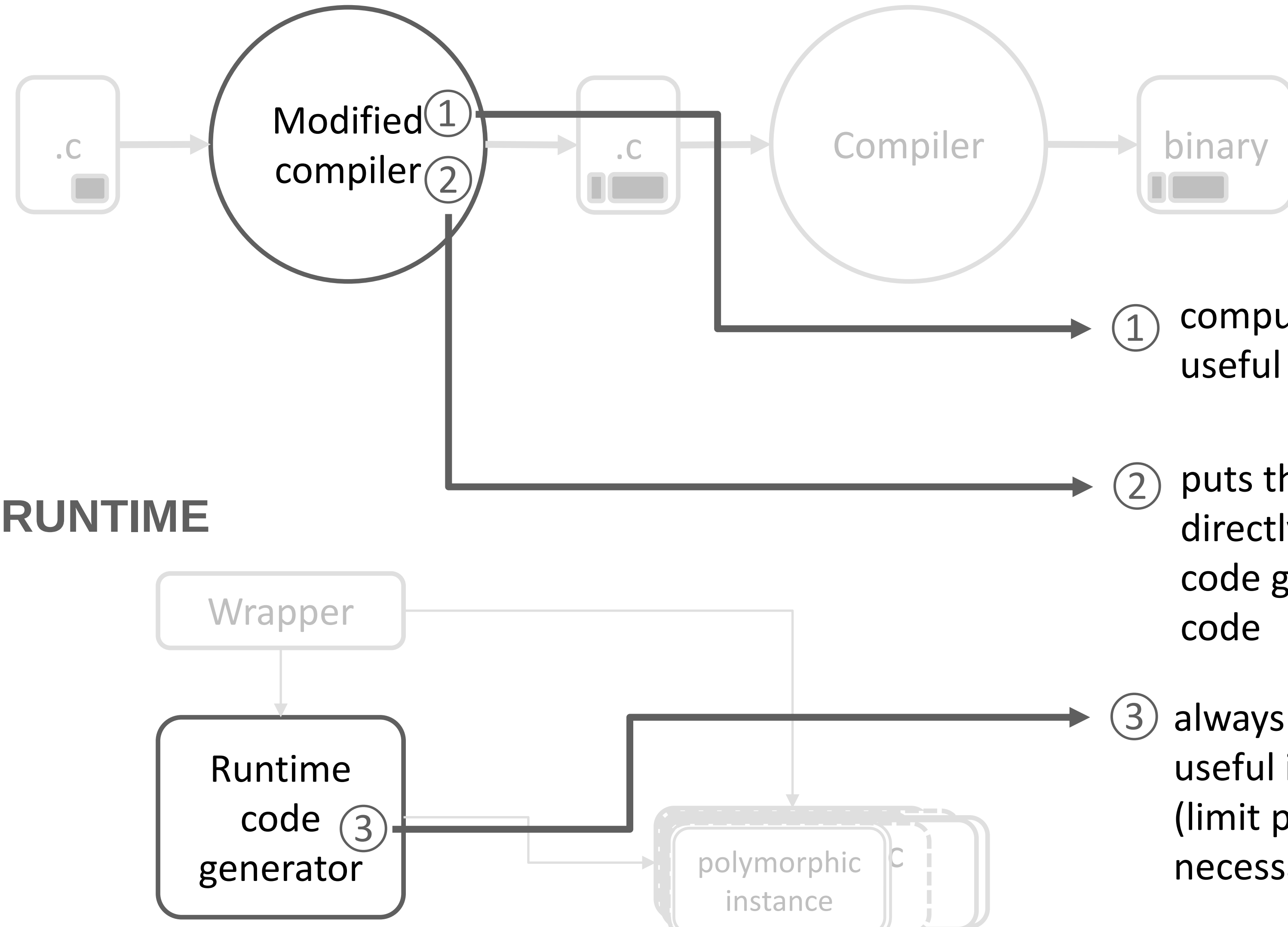
Compute a more realistic size using a threshold

threshold: probability of having an overflow
10^{-6} by default

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

OVERFLOW PREVENTION

STATICALLY



RUNTIME

1 computes the size of useful instructions

2 puts the information directly in runtime code generator's code

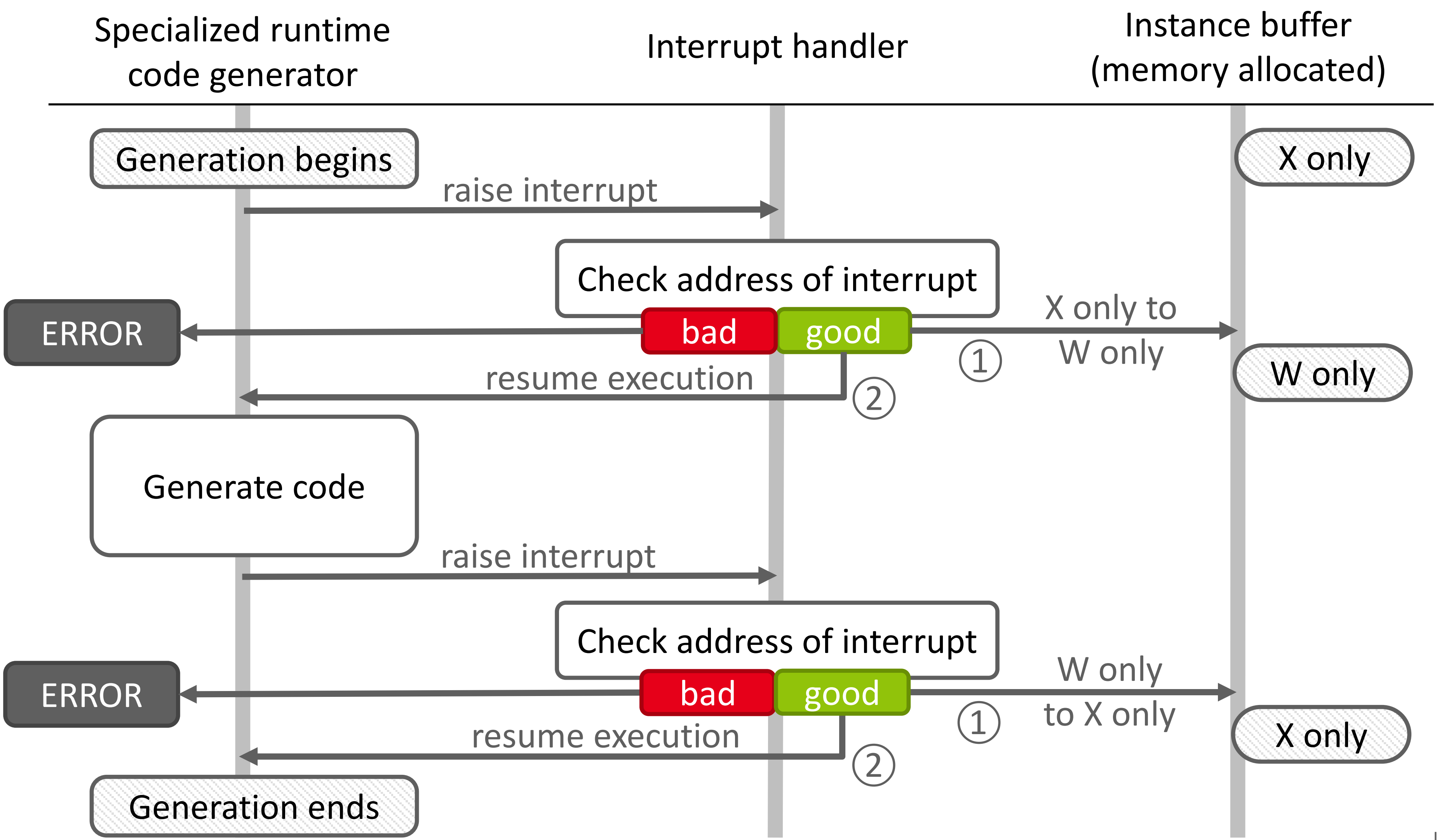
3 always keep space for useful instructions (limit polymorphism if necessary)

OUTLINE

- **Background**
  - Side channel attacks
  - Software countermeasures
  - Code polymorphism
- **Automated application of code polymorphism**
  - Overview
  - Code transformations used
  - Memory management
    - Memory allocation & overflow prevention
    - Memory permissions
- **Experimental evaluation**
  - Performance evaluation
  - Security evaluation

MANAGEMENT OF MEMORY PERMISSIONS

Objective: Guarantee W ⊕ X and that only the generator can write into the buffer

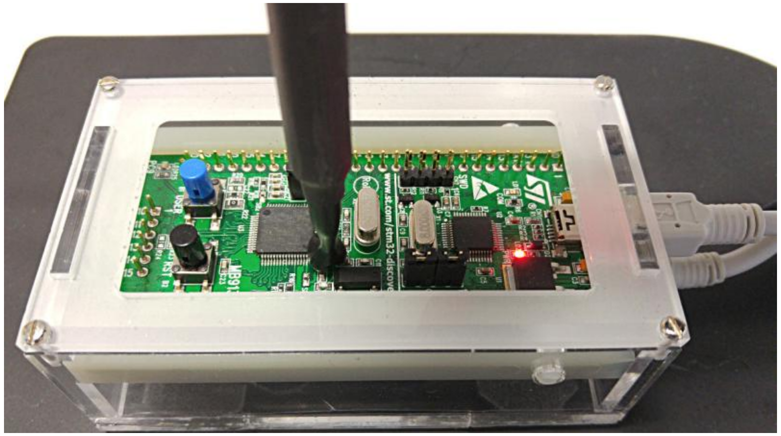


OUTLINE

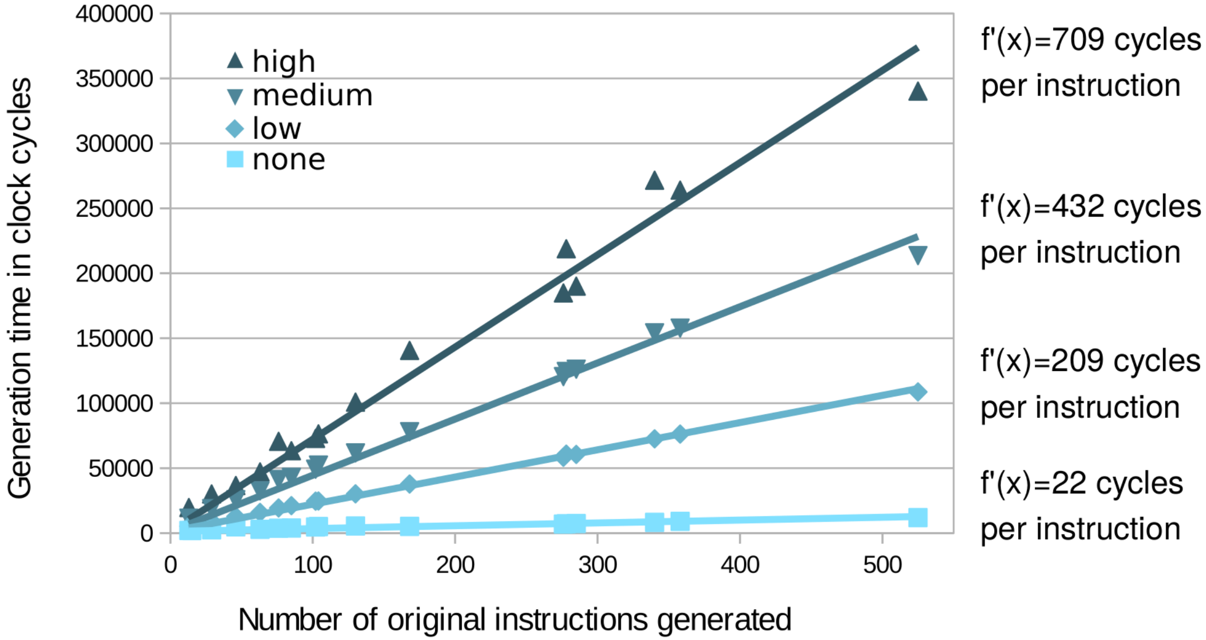
- **Background**
  - Side channel attacks
  - Software countermeasures
  - Code polymorphism
- **Automated application of code polymorphism**
  - Overview
  - Code transformations used
  - Memory management
- **Experimental evaluation**
  - Performance evaluation
  - Security evaluation

EXPERIMENTAL SETUP

- **15 different test cases**
- **4 different selected configurations**
  - none: no polymorphism
  - low: only noise instructions, generation is done every 250 executions
    - Theoretical number of variants is already very high!
      $>6\times10^{22}$ variants for a 10 instructions code
      $>10^{704}$ variants for the 278 instructions AES we use
  - medium: all transformations activated, generation is done every execution
  - high: all transformations activated, different probability model for noise instructions insertion, generation is done every execution
- **STM32 board (ARM cortex M3 – 24 MHz – 8kB of RAM)**



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



overheads depend on configuration
→ trade-off to find

generation done in linear complexity

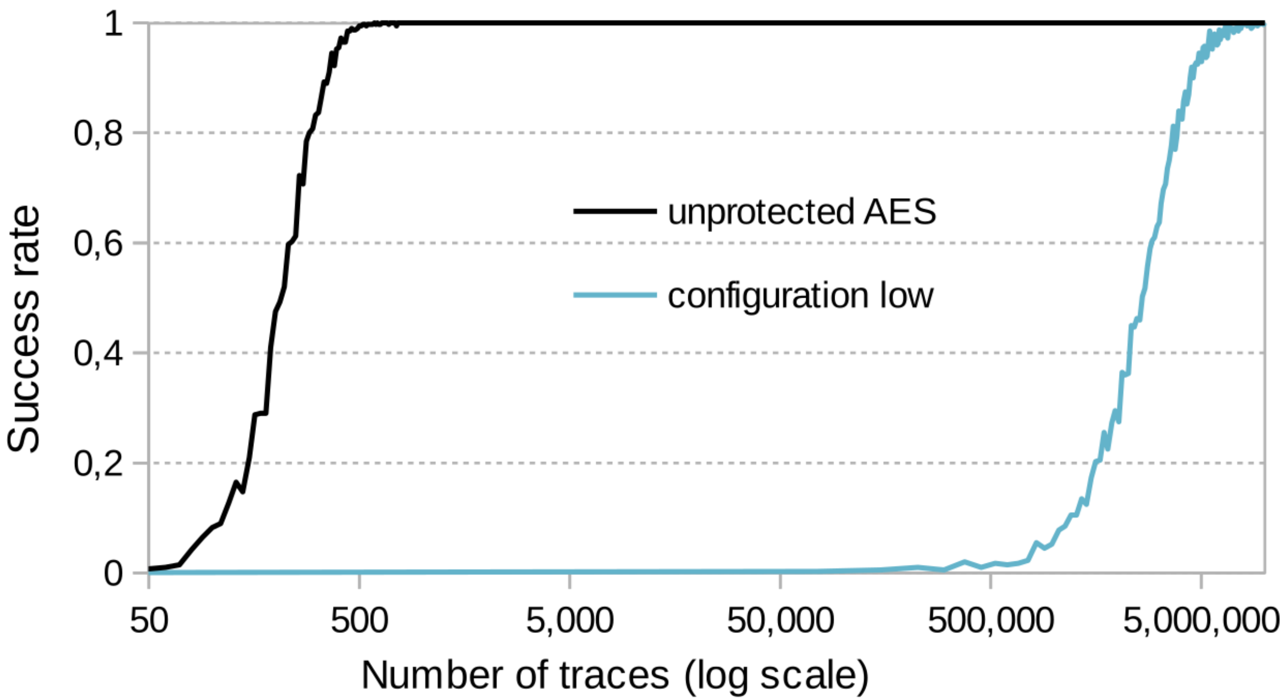
more results in our paper

- **Background**
  - Side channel attacks
  - Software countermeasures
  - Code polymorphism
- **Automated application of code polymorphism**
  - Overview
  - Code transformations used
  - Memory management
- **Experimental evaluation**
  - Performance evaluation
  - Security evaluation

SECURITY EVALUATION

Technical details:
PicoScope 2208A, EM probe RF-U 5-2 (Langer), PA 303 preamplifier (Langer)
Sampling at 500 Msample/s with 8bits resolution, 24500 samples per trace

- **CPA on Sbox output with HW**
- **Success rate at 0.8 in**
  - 290 traces for unprotected AES
  - 3 800 000 traces for configuration low



13000x more traces needed!

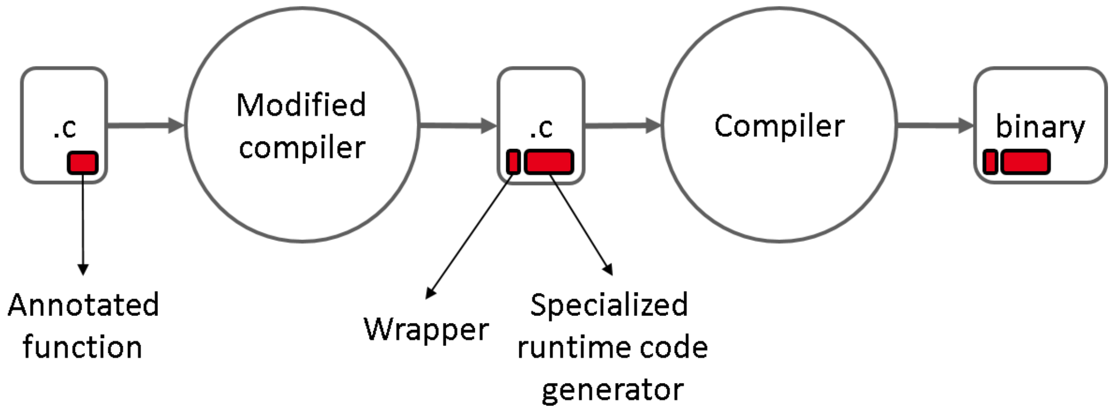
Execution time overhead: **x2.5** including generation cost!

more results in our paper

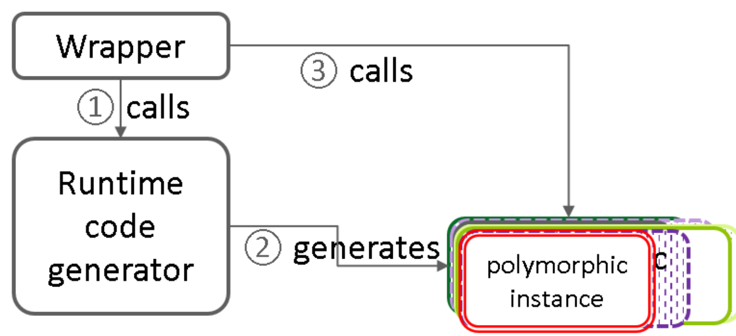
CONCLUSION

- **Automatic ⍰**
- **Configurable ⍰**
- **Efficient ⍰**
- **With static memory allocation of a realistic size ⍰**
- **With memory permission management ⍰**
- **Usable on constrained devices ⍰**
- **Open question: interest of code polymorphism against fault injection attacks?**

STATICALLY



RUNTIME



Automated software protection for the masses against side-channel attacks

Nicolas Belleville
Damien Couroussé
Karine Heydemann
Henri-Pierre Charles

This work was partially funded by the French National Research Agency (ANR) as part of the projects COGITO and PROSECCO, respectively funded by the programs INS-2013 under grant agreement ANR-13-INSE-0006-01 and AAP-2015 under grant agreement ANR-15-CE39.

Thank you for your attention

Questions?

contact:
nicolas.belleville@cea.fr

Centre de Grenoble
17 rue des Martyrs
38054 Grenoble Cedex

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