FROM RESEARCH TO INDUSTRY

cea tech

COGITO: Runtime Code Generation to Secure Devices

8emes rencontres de la communauté française de compilation – Nice

Damien Couroussé

damien.courousse@cea.fr

CEA-LIST / DACLE / LIALP – Grenoble

July 3, 2014

COGITO

www.cea.fr

leti & list

Domain

Runtime code generation

. . . for security purposes in embedded systems, mainly against physical attacks

Domain

Runtime code generation

. . . for security purposes in embedded systems, mainly against physical attacks

Problem: program code is invariant

Code polymorphism (thanks to runtime code generation) could improve this:

- reverse engineering
- physical attacks

Motivation pitch

Domain

Runtime code generation

. . . for security purposes in embedded systems, mainly against physical attacks

Problem: program code is invariant

Code polymorphism (thanks to runtime code generation) could improve this:

- reverse engineering
- physical attacks

Objectives: explore the use of runtime code generation as a means to secure embedded systems against physical attacks

How? `deGoal`:

- runtime code generation and code optimizations
- suitable for constrained embedded systems:
  - fast code generation
  - within tiny memory footprints: works on TI's Launchpad MSP430 (512 B RAM)

Outline

This talk is about:

1. An overview of security issues – aka physical security of embedded systems for dummies

   . . . and how code polymorphism is likely to bring new solutions
2. A practical solution to achieve code polymorphism for security: `deGoal`
   - overview of `deGoal`
   - modification for security purposes
   - demo time

COGITO – ANR INS 2013

- Project coordination
- Bringing the `deGoal` framework
- Compilation & runtime code generation

- Scientific coordination
- Security analysis
- Physical attacks and software countermeasures
- JavaCards

- Security analysis
- Physical attacks, HW/SW countermeasures
- Experimental validation

Public website

www.cogito-anr.fr

Outline

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

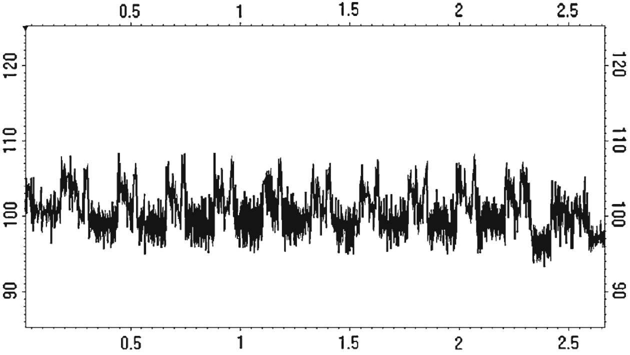
Approx. typology of physical attacks

- **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
- **Fault injection attacks**
  - under/over voltage drops
  - iom / laser beam, optical illumination
  - glitch attacks
  - ...

Approx. typology of physical attacks

- **Reverse engineering**
  - HW inspection: decapsulation, abrasion, chemical etching, memory ex
  - S
- **Side c** erential –), CPA (
  - E
  - P
  - A
  - T
- **Fault**
  - u
  - io
  - gl
  - ..

SPA on AES [Kocher, 2011]:

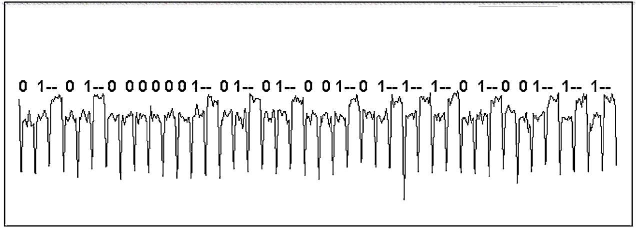


the AES rounds are "clearly" visible

Approx. typology of physical attacks

- **Reverse engineering**
  - HW inspection: decapsulation, abrasion, chemical etching, memory extraction, etc.
  - S
- **Side c** ...erential –), CPA (
  - E
  - P
  - A
  - T
- **Fault**
  - u
  - io
  - gl
  - ...

SPA on RSA [Kocher, 2011]:

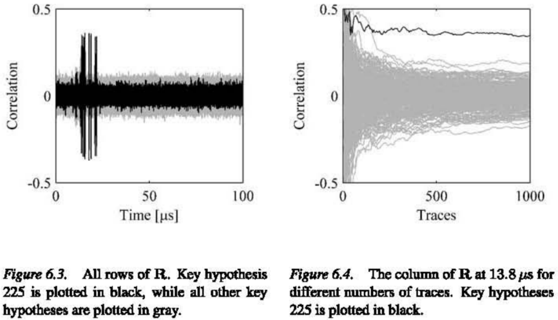


Direct access to key's contents:

- bits 0 = square
- bits 1 = square + mul

Approx. typology of physical attacks

- DPA on AES:
  1. get *n* traces from the target, using selected clear inputs
  2. compute intermediate values for each input, for each possible key values
  3. compute {power/EM/timing...} estimation from the intermediate values
  4. correlate with the measurement traces



[Mangard, 2007]

Approx. typology of physical attacks

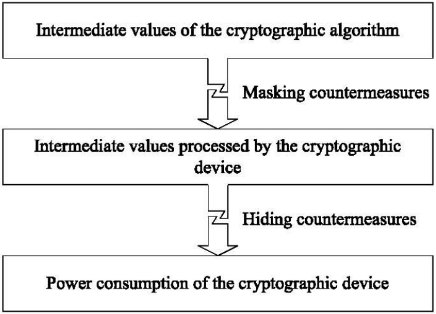
- **Reverse engineering**
  - HW inspection: decapsulation, abrasion, chemical etching, memory extraction, etc.
  - SW inspection: debug, memory dumps, code analysis, etc.
- **Side channel attacks**: SPA (Simple Power Analysis), DPA (Differential –), CPA (Correlation –)... ⇒ temporal & spatial sensitivity
  - Electromagnetic analysis
  - Power analysis
  - Acoustic analysis
  - Timing attacks
- **Fault injection attacks**
  - under/over voltage drops
  - iom / laser beam, optical illumination
  - glitch attacks
  - ...

Approx. 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 –)... ⇒ temporal & spatial sensitivity
  - Electromagnetic analysis
  - Power analysis
  - Acoustic analysis
  - Timing attacks
- **Fault injection attacks** ⇒ temporal & spatial sensitivity
  - under/over voltage drops
  - iom / laser beam, optical illumination
  - glitch attacks
  - ...

Software protections against physical attacks

Hiding and masking decorrelate data processing from power consumption



Hiding: remove the data dependency of the power consumption
Masking: randomize the intermediate values that are processed by the cryptographic device (vs. algorithmic intermediate values)

[Mangard, 2007]

Our proposal

Use **code polymorphism** to tackle the problem of **program contents** as an **invariant**

Code polymorphism

Definition

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

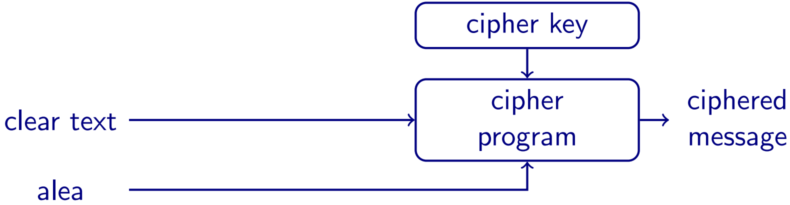
How?

- Generate secured (& polymorphic) functions at runtime
- . . . thanks to a code generator
- Generate a new morphing each time it is necessary
  - security factor ω

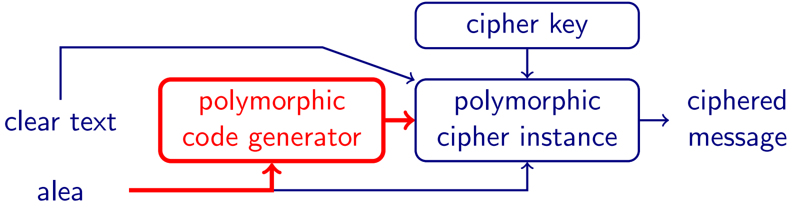
What for ?

- SW reverse: more difficult
  - the secured code is not available before runtime
  - the secured code regularly changes its form
  - meta-analysis of the code generator?
- polymorphism changes **the spatial and temporal properties** of the secured code: side channel attacks fault attacks
- combine usual SW protections against 2nd step attacks

COGITO sketched



COGITO sketched



Outline

Overview of `deGoal`

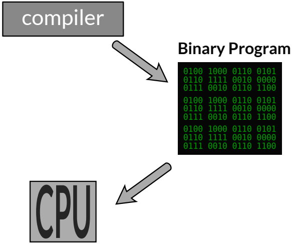
1. Program performance: strong correlation to data
2. Static compilation: no (or almost no) knowledge about the data

- `deGoal` is a tool that allows to design **compilettes**
- A compilette is:
  - an *ad hoc* code generator that targets *one* kernel ($\neq$ application)
  - aimed to be invocated at runtime

Overview of `deGoal`

1. Program performance: strong correlation to data
2. Static compilation: no (or almost no) knowledge about the data

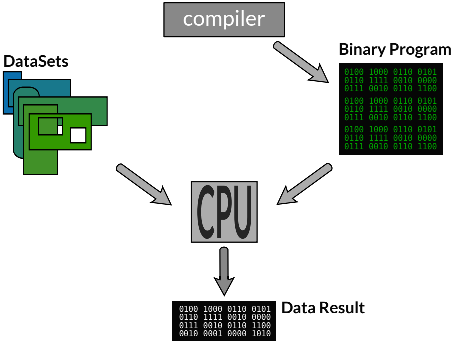
- `deGoal` is a tool that allows to design **compilettes**
- A compilette is:
  - an *ad hoc* code generator that targets *one* kernel ($\neq$ application)
  - aimed to be invocated at runtime



Overview of deGoal

1. Program performance: strong correlation to data
2. Static compilation: no (or almost no) knowledge about the data

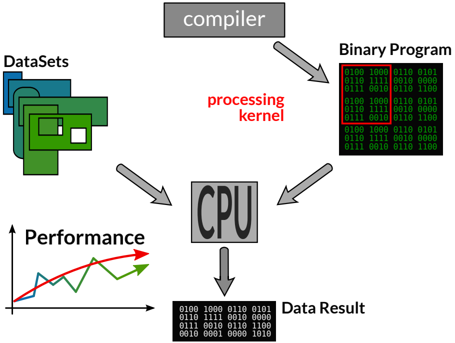
- `deGoal` is a tool that allows to design **compilettes**
- A compilette is:
  - an *ad hoc* code generator that targets *one* kernel ($\neq$ application)
  - aimed to be invocated at runtime



Overview of `deGoal`

1. Program performance: strong correlation to data
2. Static compilation: no (or almost no) knowledge about the data

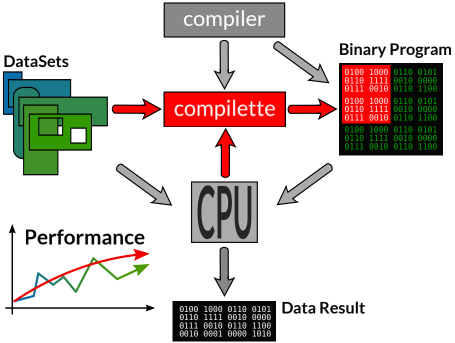
- `deGoal` is a tool that allows to design **compilettes**
- A compilette is:
  - an *ad hoc* code generator that targets *one* kernel ($\neq$ application)
  - aimed to be invocated at runtime



Overview of deGoal

1. Program performance: strong correlation to data
2. Static compilation: no (or almost no) knowledge about the data

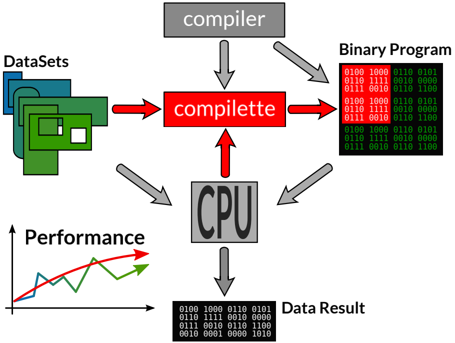
- `deGoal` is a tool that allows to design **compilettes**
- A compilette is:
  - an *ad hoc* code generator that targets *one* kernel ($\neq$ application)
  - aimed to be invocated at runtime



Overview of `deGoal`

1. Program performance: strong correlation to data
2. Static compilation: no (or almost no) knowledge about the data

- `deGoal` is a tool that allows to design **compilettes**
- A compilette is:
  - an *ad hoc* code generator that targets *one* kernel ($\neq$ application)
  - aimed to be invocated at runtime



Properties of compilettes:

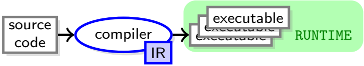
- low memory footprint
- high portability

Aim:

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

Approaches for code specialization

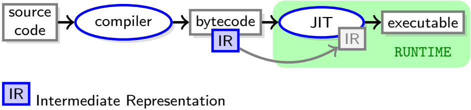
Static code versionning (e.g. C++ Templates)



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

Dynamic compilation

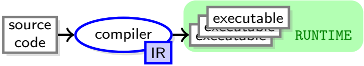
(JITs, e.g. Java Hotspot)



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

Approaches for code specialization

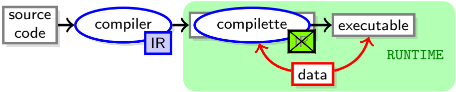
Static code versionning (e.g. C++ Templates)



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

Runtime code generation, with deGoal

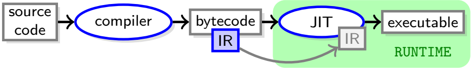
A *compilette* is an ad hoc code generator, targeting one executable



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

Dynamic compilation

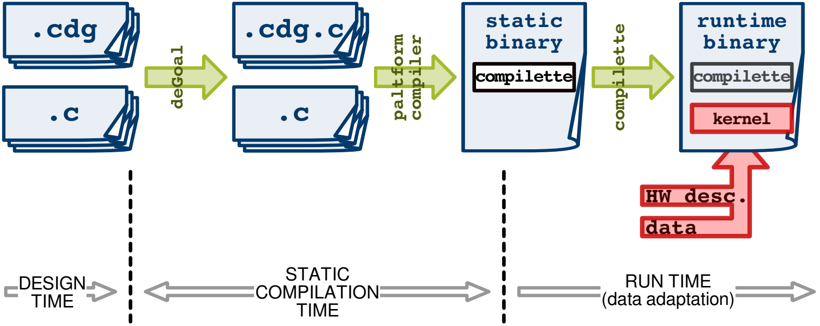
(JITs, e.g. Java Hotspot)



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

IR Intermediate Representation

Development flow using deGoal



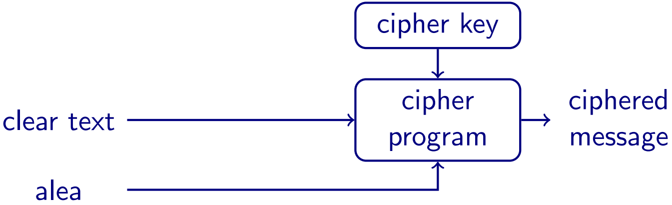
Supported architectures

- ARM 32-bits, Thumb 1 & 2 (including NEON, VFP)
  - Cortex-A8 (beagleBoard), Cortex-A9 (snowball), Cortex-M3 (STM32 discovery – 8 kB RAM)
  - gem5 + McPAT
- MSP430 from Texas Instruments
  - TI's Launchpad (512 bytes only!), Zolertia
- MIPS 32 bits
- VLIW architectures: STxP70 (ST-Microelectronics), other VLIWs under NDA
- Nvidia GPUs (Cuda PTX assembly language)

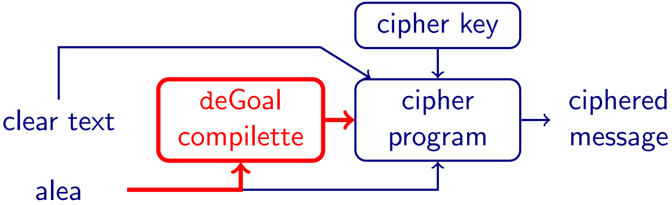
It is the only tool for dynamic code generation able to target very small processors, up to 8-bit microcontrollers

Demonstrated for the **16-bit MSP430** with only **512 bytes of RAM**: *Software Acceleration of Floating-point Multiplication using Runtime Code Generation. C. Aracil & D. Couroussé. ICEAC 2013*

A sketch of `deGoal` for COGITO



A sketch of `deGoal` for COGITO



Overview of deGoal capabilities

deGoal runtime capabilities

Performed *in this order*:

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

What does it mean for COGITO:

- 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 generation of semantically equivalent sequences
  - random mapping to physical registers [1]
  - use of semantic equivalences [2]
  - instruction scheduling [3]
  - insertion of dummy operations [3]

Potential limitations and flaws

Requirement: **writable program memory**

- Current practice:
  - generate code in RAM (most frequent case)
  - or in ROM (flash)

Potential limitations and flaws

Requirement: **writable program memory**

- Current practice:
  - generate code in RAM (most frequent case)
  - or in ROM (flash)
- **Is it acceptable for the industry of security?**

Potential limitations and flaws

Requirement: **writable program memory**

- Current practice:
  - generate code in RAM (most frequent case)
  - or in ROM (flash)
- **Is it acceptable for the industry of security?**
- **Possible workarounds?**
  - Lower the side effects of this issue:
    - obfuscate the code generator with encryption
    - ...
  - Hardware design of a dedicated block ...

Potential limitations and flaws

Requirement: **writable program memory**

- Current practice:
  - generate code in RAM (most frequent case)
  - or in ROM (flash)
- **Is it acceptable for the industry of security?**
- **Possible workarounds?**
  - Lower the side effects of this issue:
    - obfuscate the code generator with encryption
    - ...
  - Hardware design of a dedicated block ...

The **code generator** itself must be secured agains physical attacks

Out of the scope of this talk

Outline

Demo

Target : $[B] = \alpha \times [A]$

```c
typedef void (*fp)(int*);
int src[TABLE_LEN], dest[TABLE_LEN];

void vector_mul(int * A, int A_len, int alpha, int * B) {
    int i; for (i=0; i<A_len; i++) {
        B[i] = alpha * A[i];
    }
}

int main() {
    cdg_insn_t * code = CDGALLOC(ALLOC_LEN);
    compilette(code, src, vsize, alpha); /* code generation */

    fp kernel = (fp)code;
    kernel(dest);                        /* execution */

    PRINT("dest :");
    for (i = 0; i < vsize; ++i) { PRINT("%3d ", dest[i]); }
}
```

Demo

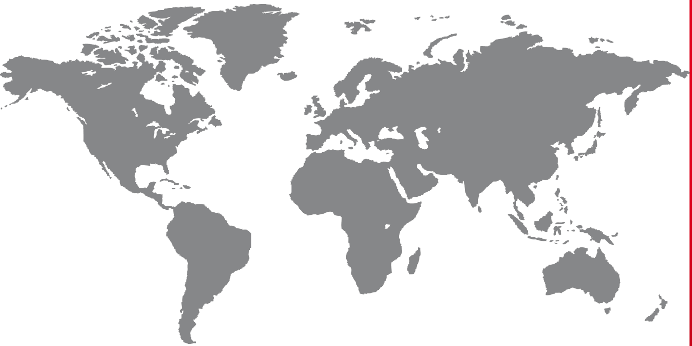
```
void compilette(cdg_insn_t* code, int * A_addr, int A_len, int alpha) {
  #[
  Begin code Prelude B_addr

  Type  ptr_t int 32
  Type  vint_t int 32 #(A_len)
  Alloc vint_t v
  Alloc ptr_t tmp

  mv tmp, #(A_addr)
  lw v, tmp
  mul v, v, #(alpha)
  sw B_addr, v
  rtn

  End
  ]#;
}
```

Demo

- non-polymorphic execution
- random register allocation
- instruction shuffling

Two positions opened !!

- Post-doc on COGITO
  keywords: security, code generation, [IoT]
- Embedded SW developper for MPSoCs
  keywords: embedded, runtime SW, code generation, parallelism



Thanks!

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

Outline

1. The COGITO project
2. Code polymorphism as a proposal to improve physical security in embedded systems
3. `deGoal`
   - Introduction to `deGoal`
   - Secured runtime code generation with `deGoal`
   - Potential limitations
4. Demo