COGITO: Runtime Code Generation to Secure Devices

8èmes rencontres de la communauté française de compilation – Nice

Damien Couroussé
damien.courousse@cea.fr

CEA-LIST / DACLE / LIALP – Grenoble

July 3, 2014
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)
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
- 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
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
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
  - ...

⇒ temporal & spatial sensitivity

SPA on AES [Kocher, 2011]: the AES rounds are “clearly” visible

SPA on RSA [Kocher, 2011]: Direct access to key’s contents:
  - bits 0 = square
  - bits 1 = square + mul

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 Power Analysis), CPA (Correlation Power Analysis).

- **Electromagnetic analysis**

- **Power analysis**

- **Acoustic analysis**

- **Timing attacks**

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

---

**SPA on AES [Kocher, 2011]:**

```
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
```

- **DPA on AES:**

- **COGITO: Runtime Code Generation to Secure Devices**

- **DACLE Division**

- **July 3, 2014**
Approx. typology of physical attacks

- **Reverse engineering**
  - HW inspection: decapsulation, abrasion, chemical etching, memory extraction, etc.
  - SPA on RSA [Kocher, 2011]:

- **Side channel**
  - CPA (Correlation –)
  - E, P, A, T

- **Fault**
  - Direct access to key’s contents:
    - bits 0 = square
    - bits 1 = square + mul
    - ...

SPA on AES [Kocher, 2011]:

The AES rounds are “clearly” visible

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

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 $\{\text{power/EM/timing} \ldots \}$ estimation from the intermediate values
4. correlate with the measurement traces

![Graph 1](image1.png)

**Figure 6.3.** All rows of $R$. Key hypothesis 225 is plotted in black, while all other key hypotheses are plotted in gray.

![Graph 2](image2.png)

**Figure 6.4.** The column of $R$ at 13.8 $\mu$s for different numbers of traces. Key hypotheses 225 is plotted in black.

[Source: 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
  - ...

SPA on AES [Kocher, 2011]: the AES rounds are "clearly" visible

SPA on RSA [Kocher, 2011]: Direct access to key's contents:
bits 0 = square
bits 1 = square + mul

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]
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)

[1] [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 $\omega$

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

- clear text
- alea
- cipher key
- cipher program
- ciphered message

COGITO: Runtime Code Generation to Secure Devices | DACLE Division | July 3, 2014 | 10
COGITO: Runtime Code Generation to Secure Devices | DACLE Division | July 3, 2014 | 10

- clear text
- alea
- polymorphic code generator
- cipher key
- polymorphic cipher instance
- ciphered message
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
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 (≠ application)
  - aimed to be invocated at runtime

```
Compiler
```
```
Binary Program
```
```
CPU
```

Properties of compilettes:
- low memory footprint
- high portability

Aim:
- Modify kernel’s binary instructions according to the input data whenever needed 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 (≠ 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 (≠ 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 (≠ 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)

Runtime code generation, with deGoal
A compilette is an ad hoc code generator, targeting one executable

Dynamic compilation
(JITs, e.g. Java Hotspot)
Development flow using deGoal

- .cdg
- .c
- .cdg.c
- .c
- deGoal
- platform compiler
- compilette
- static binary
- kernel
- runtime binary
- compilette
- HW desc.
- data
- RUN TIME (data adaptation)
- STATIC COMPILATION TIME
- DESIGN TIME

COGITO: Runtime Code Generation to Secure Devices | DACLE Division | July 3, 2014 | 14
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

- clear text
- alea
- cipher key
- cipher program
- ciphered message
A sketch of deGoal for COGITO

clear text

alea

deGoal compilette

cipher
program

cipher key

ciphered message
Overview of deGoal capabilities

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)

---

**COGITO: Runtime Code Generation to Secure Devices | DACLE Division | July 3, 2014 | 18**
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 against physical attacks

Out of the scope of this talk
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
 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]); } 
}
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 developer for MPSoCs
  keywords: embedded, runtime SW, code generation, parallelism
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