All paths lead to Rome: Polymorphic Runtime Code Generation for Embedded Systems

CS2 2018 – Manchester
2018-01-24

Damien Couroussé, Thierno Barry, Bruno Robisson, Nicolas Belleville, Philippe Jaillon, Olivier Potin, Hélène Le Bouder, Jean-Louis Lanet, Karine Heydemann

Damien Couroussé, CEA – LIST / LIALP; Grenoble Université Alpes
damien.courousse@cea.fr
CEA: French Atomic & Alternative Energies Commission

» 15 700 employees
» 10 research centers
» Budget of 3.9 billion €
» 580 patents/year
» 4000 publications/year
» 120 startup created since 1984
DACLE
Architectures, IC Design & Embedded Software Division

300 members
160 permanent researchers

60 PhD students & postdocs

> 150 scientific papers per year

45 patents per year

PhD and post-doc offers:
http://www-instn.cea.fr/formations/formation-par-la-recherche/doctorat/liste-des-sujets-de-these.html
Worldwide IoT Security Spending Forecast
(Millions of Dollars)

<table>
<thead>
<tr>
<th>Year</th>
<th>Spending</th>
</tr>
</thead>
<tbody>
<tr>
<td>2014</td>
<td>$231.86</td>
</tr>
<tr>
<td>2015</td>
<td>$281.54</td>
</tr>
<tr>
<td>2016</td>
<td>$348.32</td>
</tr>
<tr>
<td>2017</td>
<td>$433.95</td>
</tr>
<tr>
<td>2018</td>
<td>$547.20</td>
</tr>
</tbody>
</table>

Source: Gartner (April 2016)
BESTIARY OF EMBEDDED SYSTEMS

... IN NEED FOR SECURITY CAPABILITIES

Smart Card

Secure Element inside...

... And many other things

Cyber-Security

... SUPPORTED BY TOOLS
- flexibility
- automated application of protections
- deploy at large scale
One of the major threats against secure embedded systems

- The only effective class of attacks against implementations of cryptography
- Relevant in many cases against cyber-physical systems: bootloaders, firmware upgrade, reverse-engineering, user authentication, etc.

### Observation-based: side channel attacks
- hiding
- masking

### Perturbation-based: fault attacks
- tolerance
- detection
Automated application of software countermeasures against physical attacks

=> A toolchain for the compilation of secured programs

- Countermeasures supported:
  - Fault tolerance, including multiple fault injections
  - Fault detection
  - Control-Flow Integrity
    - Combined with integrity of execution paths at the granularity of a single machine instruction
  - Polymorphism

- LLVM: an industry-grade, state-of-the-art compiler (competitive with GCC)
Information leakage: information related to secret data and secret operations “sneaks” outside of the secured component (via a side channel)

Hiding: “reducing the SNR”, where

- Signal -> information leakage
- Noise -> everything else

- Temporal dispersion: spread leakage at different computation times
  - Shuffle independent operations
  - Insert «dummy» operations to randomly delay the secret computation

- Spatial dispersion:
  - Move the leaky computation at different places in the circuit
    - E.g. use different registers
  - Modify the “appearance” of information leakage
    - E.g. use different operations

In practice, a secured product combines masking and hiding countermeasures.
SOFTWARE PROTECTIONS BASED ON HIDING

The many ways to reach the same program results!

- **Insertion of dummy operations**
  - Same computation, with fake data
  - Random delays
  - Broken by recent filtering or re-synchronisation attacks

- **Multi-versionning**
  - Increases the code size,
    - also increases the attack surface
  - The path can be computed from an authentication key
  - Code traps can detect malicious usages

- **Polymorphic runtime code generation**
  - Opens the door to many code transformation opportunities, e.g. random allocation of registers
  - The protected code is not available before runtime, i.e. for static analysis
  - The main overhead comes from runtime code generation
  - Can reduce the frequency of code generation to reduce the overhead
CODE POLYMORPHISM

**Code polymorphism:** regularly changing the behavior of a (secured) component, at runtime, while maintaining unchanged its functional properties, with runtime code generation

- Protection against physical attacks: side channel & fault attacks
  - Polymorphism changes the spatial and temporal properties of the secured code
  - Can be combined with other state-of-the-Art HW & SW Countermeasures
- Implementation with runtime code generation

(patented)
Working Principle

Runtime code generation for embedded systems

Reference version:

foo.c

AES 8 bits.c

Platform compiler

Binary image

Polymorphic version, with COGITO:

foo.c

AES 8 bits.c

COGITO

AES.cdg.c

Platform compiler

Polymorphic code generation lib.

Binary image

Polymorphic code generator

rand()

Runtime code generation

Polymorphic instance of AES
AES 8-BIT. PERFORMANCE OVERHEAD

\[ k = \frac{t_{\text{gen}} + \omega \times t_{\text{poly}}}{\omega \times t_{\text{ref}}} \]

- **k**: performance overhead factor
- **\( \omega \)**: runtime code generation interval

<table>
<thead>
<tr>
<th>AddRoundKey</th>
<th>SubBytes</th>
<th>All round functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>k Min.</td>
<td>k Avg.</td>
<td>k Max.</td>
</tr>
<tr>
<td><strong>( \omega=1 )</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.16</td>
<td>4.91</td>
<td>6.37</td>
</tr>
<tr>
<td><strong>( \omega=10 )</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.32</td>
<td>1.50</td>
<td>1.66</td>
</tr>
<tr>
<td><strong>( \omega=100 )</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.09</td>
<td>1.16</td>
<td>1.22</td>
</tr>
<tr>
<td><strong>( \omega=1000 )</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.09</td>
<td>1.13</td>
<td>1.18</td>
</tr>
<tr>
<td><strong>( \omega=10000 )</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.05</td>
<td>1.12</td>
<td>1.18</td>
</tr>
</tbody>
</table>

- **Variable performance results** according to
  - Settings of the polymorphic code generator
    - model of noise insertion
  - Code is slower when executed in RAM (memory accesses)
  - Room for performance improvements
    - The non-polymorphic generated code is slower than the reference
# AES 8 BIT. MEMORY FOOTPRINT

<table>
<thead>
<tr>
<th></th>
<th>text</th>
<th>data</th>
<th>bss</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unprotected</td>
<td>4857</td>
<td>52</td>
<td>1168</td>
<td>6077</td>
</tr>
<tr>
<td>AddRoundKey only</td>
<td>8785</td>
<td>56</td>
<td>2980</td>
<td>11821</td>
</tr>
<tr>
<td>SubBytes only</td>
<td>7833</td>
<td>56</td>
<td>2980</td>
<td>10869</td>
</tr>
<tr>
<td>Full polymorphic</td>
<td>14913</td>
<td>56</td>
<td>6052</td>
<td>21021</td>
</tr>
</tbody>
</table>

x3.45
A POLYMORPHIC SUBBYTES

```c
void subBytes_compilette (
    cdg_insn_t* code, unsigned char* sbox_addr, unsigned char * state_addr)
{
    #[
    Begin code Prelude

    Type uint32 int 32
    Alloc uint32 rstate
    Alloc uint32 rsbox
    Alloc uint32 rstatei
    Alloc uint32 rsboxi
    Alloc uint32 index

    mv rstate, #((unsigned int)state_addr)
    mv rsbox, #((unsigned int)sbox_addr)

    mv index, #(16)
    loop:
        sub index, index, #(1)
        lb rstatei, rstate, index      //statei = state[i]
        lb rsboxi, rsbox, rstatei     //sboxi = sbox[statei]
        sb rstate, index, rsboxi      //state[i] = sboxi
        bneq loop, index, #(0)

    rtn
    ]#;
}
Noise instructions have no effect on the result of the program

- Parametrable model of the inserted delay ~ program execution time
  - Goal:
    - Maximize standard deviation $\sigma$
    - Minimize mean $E$
  - Can insert any instruction:
    - `nop`
    - Arithmetic (add, xor...)
    - *Memory accesses* (lw, lb, ...)
    - Power hungry instructions (mul, mac...)
    - Etc.

The leaky instruction is spread over ~300 CPU cycles

$N$: number of insertions $(E, \sigma) = f(N)$
- the noise model
- the generated code
Finding a needle in a haystack...

- Relationship between the different components of power consumption:

\[
\begin{align*}
P_{\text{total}} &= P_{\text{operations}} + P_{\text{data}} + P_{\text{noise}} \\
\text{needle} &= P_{\text{exploitable}} + P_{\text{switching\_noise}} + P_{\text{electronic\_noise}} + P_{\text{const}} \\
\text{haystack}
\end{align*}
\]

- Power signal: a static and a dynamic component.
  - Static component: power consumption of the gate states \( \rightarrow a \times \text{HW(state)} \)
  - Dynamic component: power consumption of transitions in gate states \( \rightarrow b \times \text{HD(state[i], state[i-1])} \)

- Other needles & stacks
  - Electromagnetic emissions
  - Execution time
  - Chip temperature
  - Etc.
m: plaintext → controlled by the attacker or observable
k: cipher key → unknown to the attacker

S. Mangard, E. Oswald, and T. Popp,
Impact of adding a small variability:
Visible temporal dispersion of information leakage
Correlation(EM, key hypothesis) Unprotected AES

Key found!

AES, unprotected implementation
EM traces
Attack on the output of the 1\textsuperscript{st} SBOX
Correlation(EM, key hypothesis)  Unprotected AES

Main leakage: memory read of $SBOX[m\oplus k]$

Secondary leakages, at almost every CPU cycle!

AES, unprotected implementation
EM traces
Attack on the output of the 1st SBOX
Correlation(EM, key hypothesis)

#304667

Same AES, polymorphic implementation
EM traces
Attack on the output of the 1st SBOX
IMPACT OF POLYMORPHISM ON CPA

Effect of the code generation interval

Reference implementation

Polymorphic version, code generation interval: 500

Distinguish threshold = 39 traces

Distinguish threshold = 89 traces
IMPACT OF POLYMORPHISM ON CPA

Polymorphic version
code generation interval: 20

Polymorphic version,
code generation interval: 500

Distinguish threshold > 10000 traces

Distinguish threshold = 89 traces
SECURITY EVALUATION

- CPA / DPA ... attacks do not constitute a security evaluation.
- Playing the role of the attacker is great, but the attacker
  - is focused on a potential vulnerability
  - Follows a specific attack path
- Starting from the previous attack, we could change
  - The hypothetical intermediate values: output of 1st SubBytes, output of 1st AddRoundKey, input of the 10th SubBytes...
  - The power model: Hamming Weight, Hamming Distance, no power model...
  - The distinguisher: Pearson Correlation, Mutual Information...
  - There are many other attacks!
- Our evaluation target is very “leaky” (less than 1000 traces is enough)
  - Unprotected components executed on more complex targets (i.e. ARM Cortex A9) will require 100,000 to $10^6$ traces.
  - What about attacking a counter-measure in this case?
- As a security designer, you need to cover all the possible attack passes
SECURITY EVALUATION – T-TEST

TLVA: Test Leakage Vector Assessment

- Exploit Welch’s t-test to assess the amount of information leakage
- Extract two populations of side-channel observations (traces)
- Test the null hypothesis: the two populations are not statistically distinguishable → no information leakage

\[
t = \frac{\mu_0 - \mu_1}{\sqrt{\frac{s_0^2}{n_0} + \frac{s_1^2}{n_1}}},
\]

\(t > 4.5\) → confidence of 99.999% that the null hypothesis is rejected

plaintext → \(f\) → observation
SECURITY EVALUATION – T-TEST

TLVA: Test Leakage Vector Assessment

- Exploit Welch’s t-test to assess the amount of information leakage
- Extract two populations of side-channel observations (traces)
- Test the null hypothesis: the two populations are not statistically distinguishable → no information leakage

\[
t = \frac{\mu_0 - \mu_1}{\sqrt{\frac{s_0^2}{n_0} + \frac{s_1^2}{n_1}}},
\]

\[\rightarrow \text{ confidence 0}\]
NON-SPECIFIC T-TEST

Q0: fixed input plaintext
Q1: random input plaintext

SubBytes
Polymorphism is a *hiding* countermeasure against side-channel attacks. Does not *remove* information leakage; *reduces* SNR only. The t-test is usually used to verify implementations of *masking*. With polymorphism, information leakage is sufficiently blurred such that it is *not found* in observation traces, with a confidence level of 99.999%. Configurable level of polymorphism for security-performance trade-offs.

**Non-specific t-test**

- **Reference**: 228.7
- **“Low” variability**: -5.41
- **“Medium” variability**: 4.5
- **“High” variability**: 4.5

Attack complexity increasing.
Source code
Key k

Toolchain #1

Encrypted code

Processor with decryption support

Key k

runtime code generator
Polymorphic instances

Offline
Runtime

Source code

Toolchain #2

Encrypted runtime code generator
Encrypted Polymorphic instances

Source code

Toolchain #3

Encrypted code

Processor with decryption support

Key k

runtime code generator
Polymorphic instances

Encrypted runtime code generator
Encrypted Polymorphic instances

(patented)
Resistance of polymorphism to side-channel attacks
- Hiding was shown to be an effective protection
  - ... against textbook attacks
- What about more recent attacks?

Mitigation of new vulnerabilities provided by runtime code generation
- W access to program memory! $\rightarrow$ $X \oplus W$ permissions
- JIT spraying is not applicable to our implementation

Determinism and reproducibility
- Inherent to our implementation

Debug

Functional validation

Formal verification of the secured code
Formal verification of the secured code ➔ project PROSECCO

- **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
All paths lead to Rome: Polymorphic Runtime Code Generation for Embedded Systems

CS2 2018 – Manchester
2018-01-24

Damien Couroussé, CEA – LIST / LIALP; Grenoble Université Alpes
damien.courousse@cea.fr