SIDÉ-CHANNEL ATTACKS AND SOFTWARE COUNTERMEASURES
CEA TECH DACLE

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

Digital design
Programming
Analog & MEMs
Signal processing
Imaging
Test

PhD and post-doc offers:
http://www-instn.cea.fr/formations/formation-par-la-recherche/doctorat/liste-des-sujets-de-these.html

©2017 CEA France. All rights reserved | 19 July, 2019 | 3
Foot-Shooting
Prevention Agreement

I, __________, promise that once
Your Name
I see how simple AES really is, I will
not implement it in production code
even though it would be really fun.

This agreement shall be in effect
until the undersigned creates a
meaningful interpretive dance that
compares and contrasts cache-based,
timing, and other side channel attacks
and their countermeasures.

Signature ________________________
Date ________________________

http://www.moserware.com/2009/09/stick-figure-guide-to-advanced.html  – licensed them under the Creative Commons Attribution License
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

AUTOMATED APPLICATION OF COUNTERMEASURES WITH A COMPILER

Source code

- Access to program’s semantics (e.g. secret variable)
- Security properties are not guaranteed, post compilation
- Corollary: can lead to bigger overheads

Source to source approach

Compiler

← our approach

- Access to program semantics
- Control over machine code
- Benefit from compiler optimisations
- Implementation within the compiler is difficult
- Focus on generic countermeasures

Assembly approach

Binary code

- Naturally fits to low-level / machine code protection schemes
- (Re-)construction of a program representation is difficult
- Mostly ad hoc protection schemes
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)
Objective: the program is not perturbed by the injection of faults

- Countermeasure based on a protection scheme **formally verified for the ARM architecture**, against instruction skips [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]


The image contains a slide about securing and verifying programs. The slide outlines three main points:

- **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 with respect to a security model.

The slide also includes a diagram illustrating the process of compiling code with annotations for security features into secure machine code and verifying its security properties. The diagram shows the steps from the original code to the formal verification of security properties.

The text on the slide reads: "On-going joint work with LIP6, Paris (PROSECCO – ANR 2015)"
SIDE-CHANNEL ATTACKS
Physical attacks are considered (by software hackers) as not practical

- Require dedicated HW attack benches, can be quite expensive, especially for fault injection (e.g., laser benches)
- We also find low cost ones
  - E.g. The ChipWhisperer, starting at ~ 300€
- Require human expertise, but not more than other attacks

https://newae.com/tools/chipwhisperer
IoT Goes Nuclear: Creating a ZigBee Chain Reaction

“Adjacent IoT devices will infect each other with a worm that will rapidly spread over large areas”

- Philips Hue Smart lamp
  - ZigBee protocol
- Uploading malicious firmware with OTA update
  - Discovered the hex command code for OTA update
  - Firmware is protected with a single global key! Using symmetric crypto (AES-CCM).
- Attack path
  - Get access to the key ➔ side-channel attack with power analysis
  - Sign a malicious firmware
  - Take over bulbs by: plugging a bulb, war-driving around in a car, war-flying with a drone
  - Request OTA update
  - The malicious firmware can request OTA update to its neighbours to spread.

The most comprehensive book about side-channel attacks

- Excellent introduction to side-channel attacks
- Published in 2007: does not cover recent attacks and countermeasures, but still really useful

SIMPLE POWER ANALYSIS (SPA)

Direct interpretation of power consumption measurements
Extraction of information by inspection of single side-channel traces

SPA on AES [Kocher, 2011]

The AES rounds are « clearly » visible

- Nature of the algorithm
- Structure of the algorithm
  - Number of executions
  - Number of iterations
  - Number of sub-functions
  - Nature of instructions executed (memory accesses...)
- Etc.

Illustration of SPA in the wild: C. O’Flynn, “A Lightbulb Worm? A teardown of the Philips Hue.,” presented at the Black Hat, 2016. cf. slides ~60 to 70


© 2017 Damien Couroussé, CEA France. Distributed under CC Attribution License | 21 July, 2017 | 14
SIMPLE POWER ANALYSIS (SPA)

SPA on RSA [Kocher, 2011]

-- Computing \( c = b^e \mod m \)
-- Source: [https://en.wikipedia.org/wiki/Modular_exponentiation](https://en.wikipedia.org/wiki/Modular_exponentiation)

```python
function modular_pow(base, exponent, m):
    if modulus = 1 then return 0
    Assert :: (m - 1) * (m - 1) does not overflow base
    result := 1
    base := base mod m
    while exponent > 0
        if (exponent mod 2 == 1):
            result := (result * base) mod m
        exponent := exponent >> 1
        base := (base * base) mod m
    return result
```

Direct access to key contents:
- bit 0 = square
- bit 1 = square, multiply
Finding a needle in a haystack...

- Relationship between the different components of power consumption [DPA book]:

\[
P_{\text{total}} = P_{\text{operations}} + P_{\text{data}} + P_{\text{noise}}
\]

\[
P_{\text{total}} = P_{\text{exploitable}} + P_{\text{switching.noise}} + P_{\text{electronic.noise}} + P_{\text{const}}
\]  

[© 2017 Damien Couroussé, CEA France. Distributed under CC Attribution License | 21 July, 2017]

- 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.
The AES key is fixed

- A GPIO trigger is used to facilitate the trace measurements

- The attacker
  - either knows public data: plaintexts if decrypting, ciphertexts if encrypting
  - or controls the encryption/decryption function

- Text chosen attack:
  - Generate D random plaintexts
  - Ask the cipher text to the target
  - Record the EM trace during encryption

- Do the computation analysis!

Target: STM32 – ARM Cortex-M3 @ 24MHz, 128KB flash, 8KB RAM

Target: ARM Cortex-M3 @ 24MHz, 128KB flash, 8KB RAM
m: plaintext -> controlled by the attacker or observable
k: cipher key -> unknown to the attacker

observations

plaintexts

| D texts |

cryptographic algorithm

Hypothetic interm. values

| D x K |

power model

hyp. power values

| D x K |

Correlation, Pearson ...

statistical analysis

Analysis results

| K x T |

| D traces x T points |

| D x K |

Hamming distance, Hamming weight...

key hypothesis

| K |
Success rate: success probability of a successful attack

\[ SR = \Pr[ A(E_{k_0}, L) = k_0 ] \]

- \( A \): side-channel attack
- \( k_0 \): correct key
- \( E_{k_0} \): encryption with correct key
- \( L \): leakage

n-order success rate?
PGE. Partial guessing entropy

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
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

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

\[ t > 4.5 \rightarrow \text{confidence of 99.999\% that the null hypothesis is rejected} \]


Q0: fixed input plaintext
Q1: random input plaintext

SubBytes
Mono-bit t-test:
\[ Q_0 = \{ T_i \mid \text{target bit}(D_i) = 0 \}, \quad Q_1 = \{ T_i \mid \text{target bit}(D_i) = 1 \}. \]

Bytewise t-test:
\[ Q_0 = \{ T_i \mid \text{target byte}(D_i) = x \}, \quad Q_1 = \{ T_i \mid \text{target byte}(D_i) \neq x \}. \]

Number of measurements for a security evaluation with a specific t-test?

Assuming two populations of N traces \( P_i \) and \( Q_j \),

- N sufficiently large w.r.t. t-test criteria
- Leakage exhibited on P, Q by a specific or non-specific t-test
- Also assuming that information leakage is not spread over several consecutive samples

Create two new populations of traces such that:

- \( P'_0 = P_0 \)
- \( P'_i[0; i - 1] \leftarrow 0, \; i \neq 0 \)
- \( P'_i[i; ...] \leftarrow P_i[0; ...], \; i \neq 0 \)
- Similar construction for \( Q \)
Signal-to-noise ratio

\[
P_{\text{total}}^\alpha = P_{\text{operat}}^i + P_{\text{dat}}^r + a^s P_{\text{noi}}^s + K
\]

\[
P_{\text{tot}}^R = P_{\text{exploi}}^r + P_{\text{sw.noi}}^r + P_{\text{el.noi}}^t
\]

\[
SN = \frac{\text{Va} \text{r}(\text{sign}^\alpha)}{\text{Va} \text{ (noi)}} = \frac{\text{rVa} \text{ (P_{\text{exploi}}^s)}}{\text{Va} \text{ (P_{\text{sw.noi}}^t + P_{\text{el.noi}}^t)}}
\]  

[DPA book]

- Powerful metric, less sensitive to the modus operandi
- Still depends on the attacker’s model

NICV (normalised inter-class variance)

\[
SN = \frac{\text{Va} \text{ (E}(L|X))}{\text{Va} \text{ (L)}} = \frac{\text{AR}}{1 + \frac{1}{SN}}
\]

X: known plaintext or ciphertext data; L: observations

COUNTER-MEASURES AGAINST SIDE-CHANNEL ATTACKS

MASKING & HIDING
In a masked implementation, each intermediate value $v$ is concealed by a random value $m$ that is called mask: $V_m = v \cdot m$. The mask $m$ is generated internally, i.e. inside the cryptographic device, and varies from execution to execution. Hence, it is not known by the attacker.

Objective: each masked variable is statistically independent of the secret $v$.

Masking countermeasures are applied at the algorithmic level.
Our following discussions will be based on the parallel implementation of a masking scheme such as described in [2]. More precisely, we will consider the simplest example where all the shares are in \( \mathbb{GF}(2) \) (generalizations to fields follow naturally). In this setting, we have a sensitive variable \( x \) that is split into \( m \) shares such that \( x = x_1 \oplus x_2 \oplus \ldots \oplus x_m \), with \( \oplus \) the bitwise XOR. The first \( m - 1 \) shares are picked up uniformly at random: \( (x_1, x_2, \ldots, x_{m-1}) \overset{\text{R}}{\leftarrow} \{0, 1\} \), and the last one is computed as \( x_m = x \oplus x_1 \oplus x_2 \oplus \ldots \oplus x_{m-1} \).

Denoting the vector of shares \( (x_1, x_2, \ldots, x_m) \) as \( \bar{x} \), we will consider an adversary who observes a single leakage sample corresponding to the parallel manipulation of these shares. A simple model for this setting is to assume this sample to be a linear combination of the shares, namely:

\[
L_1(\bar{x}) = \left( \sum_{i=1}^{m} \alpha_i \cdot x_i \right) + N,
\]

The goal of hiding countermeasures is to make the **power consumption** of cryptographic devices *independent of the intermediate values* and *independent of the operations* that are performed. There are essentially two approaches to achieve this independence.

1. **the power consumption is random.**
2. **consume an equal amount of power** for all operations and for all data values.

Hiding countermeasures aim at **breaking the observable relation** between the **algorithm** (operations and intermediate variables) and **observations**.
**Hiding**

**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 -> part of the observations containing useful data (« 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.**
CODE POLYMORPHISM
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:

foo.c
AES.c

Polymorphic version, with COGITO:

foo.c
AES.c

Optimizations
Optimizations

© CEA 2018. All rights reserved
CODE TRANSFORMATIONS USED AT RUNTIME

<table>
<thead>
<tr>
<th>Register shuffling</th>
<th>Instruction shuffling</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>RANDOM</strong> general purpose register permutation</td>
<td>independent instructions are emitted in a <strong>RANDOM</strong> order</td>
</tr>
<tr>
<td>r4 → r11</td>
<td>xor r6, r5, r8 add r4, r4, r5</td>
</tr>
<tr>
<td>add r11, r11, r7 xor r8, r7, r5</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Semantic variants</th>
<th>Noise instructions</th>
<th>Dynamic noise</th>
</tr>
</thead>
<tbody>
<tr>
<td>replacement of an instruction by a <strong>RANDOMLY</strong> selected semantic variant</td>
<td>insertion of a <strong>RANDOM</strong> number of <strong>RANDOMLY</strong> chosen noise instructions</td>
<td><strong>RANDOM</strong> insertion of noise instructions with a <strong>RANDOM</strong> jump</td>
</tr>
<tr>
<td>add r4, r4, r5 xor r6, r5, #12348 xor r6, r6, r8 xor r6, r6, #12348</td>
<td>add r4, r4, r5 sub r7, r6, r2 load r3, r10, #53 xor r6, r5, r8</td>
<td>add r4, r4, r5 jump 0, 1 or 2 instructions sub r7, r6, r2 load r3, r10, #53 xor r6, r5, r8</td>
</tr>
</tbody>
</table>

useless instructions
**CODE TRANSFORMATIONS USED AT RUNTIME**

<table>
<thead>
<tr>
<th>Register shuffling</th>
<th>Instruction shuffling</th>
</tr>
</thead>
<tbody>
<tr>
<td>RANDOM general purpose register permutation</td>
<td>independent instructions are emitted in a RANDOM order</td>
</tr>
</tbody>
</table>

Illustration of the interactions between runtime code transformantions

**Semantic variants**
replacement of an instruction by a RANDOMLY selected semantic variant

xor r6, r5, #12348
add r4, r4, r5
xor r6, r6, r8
xor r6, r6, #12348

**Noise instructions**
insertion of a RANDOM number of RANDOMLY chosen noise instructions

sub r7, r6, r2
load r3, r10, #53
add r4, r4, r5
xor r6, r5, r8

<table>
<thead>
<tr>
<th>Dynamic noise</th>
</tr>
</thead>
<tbody>
<tr>
<td>RANDOM insertion of noise instructions with a RANDOM jump</td>
</tr>
</tbody>
</table>

add r4, r4, r5
jump 0, 1 or 2 instructions
sub r7, r6, r2
load r3, r10, #53
xor r6, r5, r8
# TOOLCHAIN CONFIGURABILITY

<table>
<thead>
<tr>
<th>Period of regeneration</th>
<th>Register shuffling</th>
<th>Instruction shuffling</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\mathbb{N}$ ( + custom regeneration policies)</td>
<td>${0, 1}$</td>
<td>${0, 1}$</td>
</tr>
</tbody>
</table>

Total configuration space:
$\{0, 1\}^2 \times \{0, 1, 2\}^2 \times \mathbb{R} \times \mathbb{N}^3$

<table>
<thead>
<tr>
<th>Semantic variants</th>
<th>Noise instructions</th>
<th>Dynamic noise</th>
</tr>
</thead>
<tbody>
<tr>
<td>${0, 1, 2}$ ( + custom extensions)</td>
<td>${0, 1, 2} \times \mathbb{R} \times \mathbb{N}$ ( + custom probabilistic models)</td>
<td>$\mathbb{N}$ ( + custom noise sequences)</td>
</tr>
</tbody>
</table>

A huge number of polymorphic variants:
- 10 original machine instructions $\rightarrow 6 \times 10^{42}$ variants
- **AES with 278 machine instructions** $\rightarrow 10^{27}$ variants (pessimist bound)
- 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)

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

Polymorphism is a hiding countermeasure against side-channel attacks
- Does not remove information leakage; reduces SNR only
- However, 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

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 le 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*)

![Graph showing execution time overhead and size overhead for various components with different levels of polymorphism.](image)
### PERFORMANCE EVALUATION OF RUNTIME CODE GENERATION

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Execution time overhead (geo. mean)</th>
<th>Size overhead (geo. mean)</th>
</tr>
</thead>
<tbody>
<tr>
<td>none</td>
<td>x1.40</td>
<td>x1.70</td>
</tr>
<tr>
<td>low</td>
<td>x2.31</td>
<td>x2.87</td>
</tr>
<tr>
<td>medium</td>
<td>x2.45</td>
<td>x3.44</td>
</tr>
<tr>
<td>high</td>
<td>x4.03</td>
<td>x3.81</td>
</tr>
</tbody>
</table>

- Overheads depend on the selected configuration → trade-off w.r.t. the expected security level
- Runtime code generation achieved in linear complexity

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:

```
foo.c

AES.c

COGITO compiler

AES.odo.c

Platform compiler

Polymorphic code generation lib.

Binary image
  Polymorphic AES code generator

rand()

Runtime code generation

Polymorphic instance of AES
```
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 (configurable)
- Computation of the code size done automatically by the compiler

Range where an overflow is possible

Size of the program buffer

Worst case is beyond system memory capacity

For a 100 instructions code (low config.), allocated size is \(5x\) smaller than worst case!
PREVENTION OF CODE BUFFER OVERFLOWS

**STATICALLY**

1. COGITO compiler
2. Platform compiler
3. binary

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)

**RUNTIME**

Wrapper

Runtime code generator

polymorphic instance

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

Specialized runtime code generator

Generation begins

Emit code

Instance buffer (memory allocated)

Interrupt handler

Check address of interrupt

bad

good

raise interrupt

resume execution

X only

W only

X only

W only

W only

X only

Check address of interrupt

bad

good

raise interrupt

resume execution

To ensure $W \oplus X$, the following steps should be taken:

1. Generation begins, and the code generator is initialized.
2. The interrupt handler is called, and the address of the interrupt is checked.
3. If the address is good, the code is emitted; if bad, an error is raised.
4. The execution is resumed, and the process repeats until the buffer is filled.

This process ensures that only the generator can write into the buffer, maintaining security and integrity.
• Leverage the compiler to implement counter-measures
  • Automation, flexibility, configurability
• Leverage compiler analysis and compiler optimisations to improve the effectiveness of counter-measures
• Verification tools are needed after the compilation of countermeasures

Ongoing directions
• Hardware security with software-only counter-measures is impossible challenging
  • Challenge your threat model
  • HW/SW co-design of countermeasures
SIDE-CHANNEL ATTACKS AND SOFTWARE COUNTERMEASURES

Damien Couroussé | CEA / LIST / DACLE
SILM Summer School – Rennes, 2019-07-12

damien.courousse@cea.fr
Fault models, at the Instruction Set Architecture (ISA) level:

1. Data alteration, down to the bit level.
   - ROM / RAM, processor registers
   - Bit flip, bit stuck-at
   - Typically: modification of loop counters, crypto data, opcode corruption.

2. Instruction skip, instruction modification
   - Typically: NOP execution, arbitrary jumps

3. Modification of the control flow, e.g., test inversion

PROGRAM ENCRYPTION

Hardware support

Compiler support

POLYMORPHIC EXECUTION OF ENCRYPTED PROGRAMS

Hardware support

Compiler support

POLYMORPHIC EXECUTION OF ENCRYPTED PROGRAMS

Toolchain support

- Our toolchain now targets RISCV ISA (+ARMv7)
- `llvm-RISCV` back-end generating encryption-ready binaries
- Standalone `binary encryptor`
- HW decryption added to the Spike Instruction Set Simulator
- Currently working on the encryption of polymorphic instances
AES, TIME AFTER TIME (BUT SO USEFUL...)

Encryption Process:

- Plaintext
- AddRoundKey
- SubBytes
- ShiftRows
- MixColumns
- AddRoundKey

Decryption Process:

- Plaintext
- AddRoundKey
- InvSubBytes
- InvShiftRows
- InvMixColumns
- InvSubBytes
- InvShiftRows
- AddRoundKey

© 2017 Damien Couroussé, CEA France. Distributed under CC Attribution License | 21 July, 2017 | 55
STANDARD COMPILERS AND SECURITY
• **Duties:** assurance of functional equivalence between source code and machine code
  - “functional” / “functionality” is usually not precisely defined
    - Side effects?
    - Determinism of time behaviour? (real time execution)
    - Undefined behaviours?
  - No formal assurance
    - Except few works, such as CompCert
  - Correctness by construction?
    - The source code written by the developer is not always valid

• **Objectives:** optimise one or several performance criteria
  - Execution time
  - Resources: e.g. memory consumption
  - Energy consumption, power consumption
  - There is no complete criterion for optimality, and no convergence
    - Nature of the algorithm used
    - Relation to architecture / micro-architecture
    - Data…
• Rights
  • Reorganise contents of the target program, as long as program semantics is preserved
    • Machine instructions, basic blocs
  • Select the best translation for a source code operation / instruction
  • Remove parts of the program, as long as the program functionality is considered to be preserved (i.e. the computation does not participate in producing the program results)

• Some classical optimisation passes:
  • dead code elimination
  • global value numbering
  • common-subexpression elimination
  • strength reduction
  • loop strength reduction, loop simplification, loop-invariant code motion

• LLVM’s Analysis and Transform Passes, the 2016/06/30
  • 40 analysis passes
  • 56 transformation/optimisation passes
  • 10 utilitary passes
  • … backends, etc.
EXPLOITATION OF SIDE-CHANNEL INFORMATION LEAKAGE

Simple power analysis (SPA)

SPA leaks from an RSA implementation

Correlation Power/EM Analysis (CPA/CEMA) – Can be generalised to any physical observation of the secured computation

• AES, unprotected implementation
• EM traces
• Attack on the output of the 1st SBOX

Key found!

After the encryption of 4240 Bytes of data!
EXPLOITATION OF SIDE-CHANNEL INFORMATION LEAKAGE

Simple power analysis (SPA)

SPA leaks from an RSA implementation

Correlation Power/EM Analysis (CPA/CEMA) – Can be generalised to any physical observation of the secured computation

Main leakage: memory read of SBOX[m⊕k]
Secondary leakages, at almost every CPU cycle!

- AES, unprotected implementation
- EM traces
- Attack on the output of the 1st SBOX