# Compiler Techniques for Security

Alessandro Barenghi

Dipartimento di Elettronica e Informazione e Bioingegneria (DEIB)
Politecnico di Milano

*alessandro.barenghi - at - polimi.it*

April 15, 2015

# Overview

## Lesson contents

- The typical use of a compiler is to translate high level code $\rightarrow$ machine executable code
- To this end, the compiler employs frameworks to systematically analyze the code
- One of the most employed ones is DataFlow Analysis, which you have already seen
- DataFlow Analysis (DFA) can be exploited to detect at compile time security vulnerabilities

# Security issues in cryptography

## Common attacks

- In this lesson we will tackle security issues with symmetric cryptographic algorithms
- Such algorithms mix together the known input (plaintext) and a secret key to yield a non intelligible ciphertext
- It is unfeasible to retrieve the key from plaintexts and ciphertexts only
- Common attacks to these algorithms try to break them "at-the-ends" i.e. only employing the mathematical structure of the algorithm
- Side Channel Attacks (SCAs) take into account the fact that the algorithms are actually run on a device

# Attack techniques

## Taxonomy

Side Channel Attacks (SCAs) are split into two categories:

- **Passive SCAs**: These attacks rely on observing the device during its regular functioning, recording environmental parameters
  - The correlation between the recorded parameter and the values being computed is exploited
- Typical environmental parameters: power consumption, EM emissions
- **Active SCAs**: These attacks disturb the correct execution of a cipher, leading to an erroneous result.
  - The correlation between a correct and a wrong ctx, or the fact itself that the computation failed are exploited

# Attack techniques

## Passive SCAs

- We will use as a case study the power consumption side channel
- Key idea: the power consumption of a computing device depends on the values being computed
- The same ideas can be applied to other different side channels, with the proper adaptations
- The information leaked on the side channel can be exploited with different methods:
  - **Simple Power Analysis (SPA)**: The attacker exploits the key-dependencies in the control flow of the algorithm
  - **Differential Power Analysis (DPA)** and derivates: The attacker exploits the key-dependencies in the data flow of the algorithm
- Both methods rely on measuring the power consumption of the device during a cipher computation

# Differential Power Attacks

## General Workflow

- To perform Differential Power Analysis (and derived ones) we focus on data-dependent power consumption variations
- The first step is to measure the side-channel while a computational operation involving a small part of the key is performed
- The attacker separately makes an hypothesis on the value of the measurement, guessing the value of the part of the key (i.e. one hypothesis per possible value)
- The attacker checks which hypothesis predicts correctly the measurement and obtains the key part

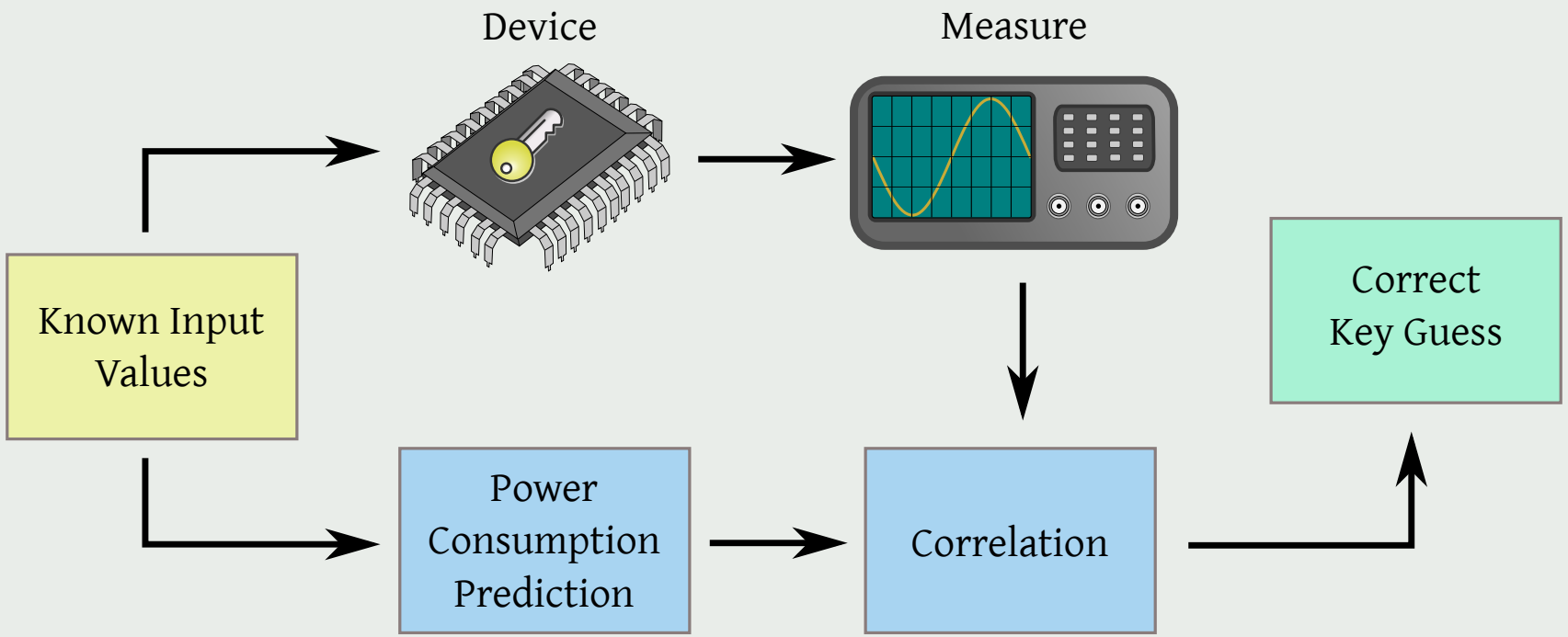
# Differential Power Attacks

## Simplifying assumptions

- We will, at first, do some simplifying assumptions for clarity's sake
- In practice, there is a way to deal with them all
- Assume thus that :
  1. **Perfectly Timed** We know exactly the time instant when every operation of the cipher is performed
  2. **Noise Free** We are able to measure perfectly the power consumption of the portion of the device performing the actual encryption
  3. **White Box** We know perfectly the implementation, down to the single logic gate/assembly instruction level.

# Overview

## Differential Power Attack workflow



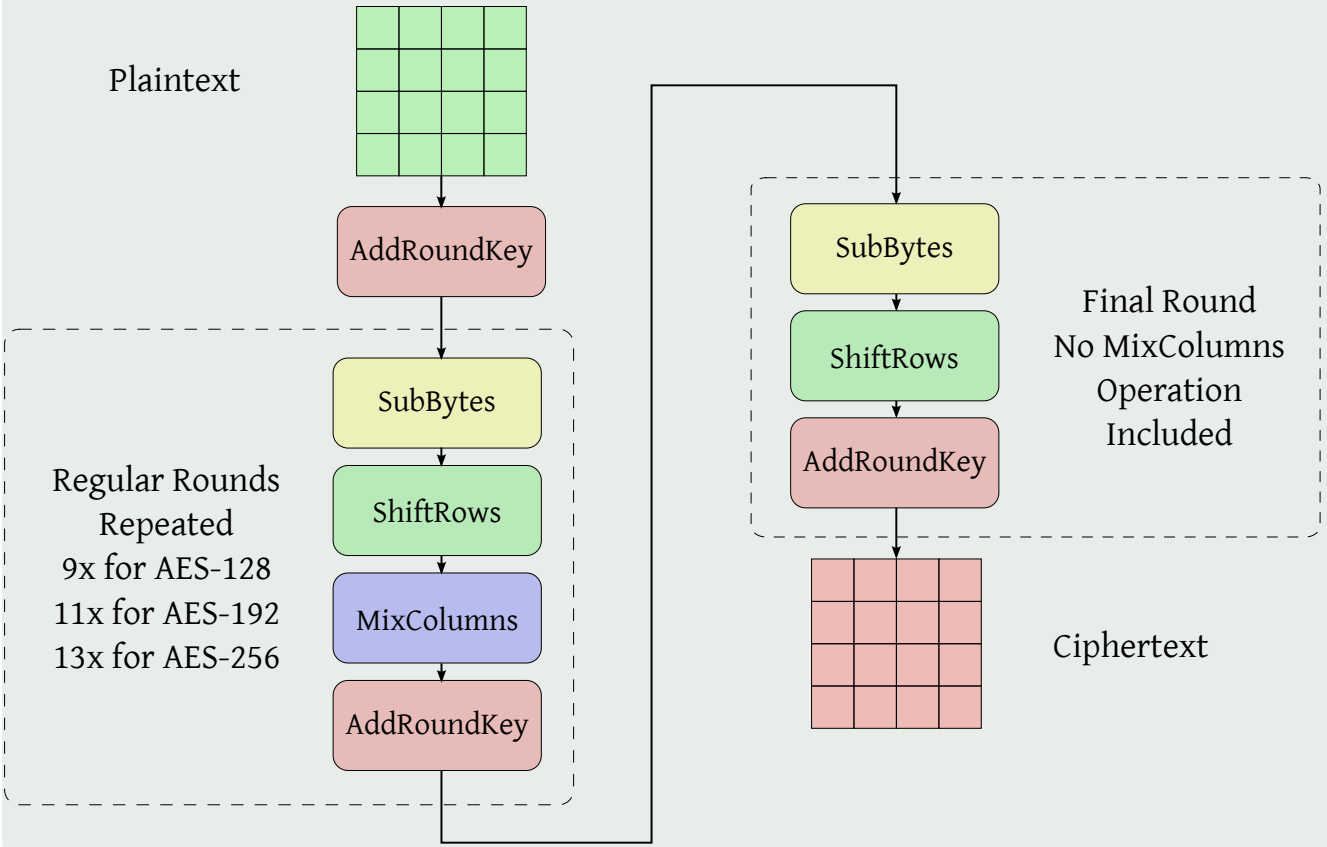
# Hypotheses on intermediate values

## Predicting the power consumption

- The only way for an attacker to predict the data dependent part of the power consumption is to know the computed data
- The attacker does not know the secret key, he's forced to guess its value, but this would imply a full-scale bruteforce attack
- Key Idea: the secret key is directly employed through small independent computations (f.i. bitwise XORs)
- Model only a small part of the key-involving operations, say, 8 bits
- Compute a hypothetical value for the data depending on the guessed 8-bit key value and the known input to the device
- The result is a list of 8-bit hypothetical values, each one relying on a different key guess

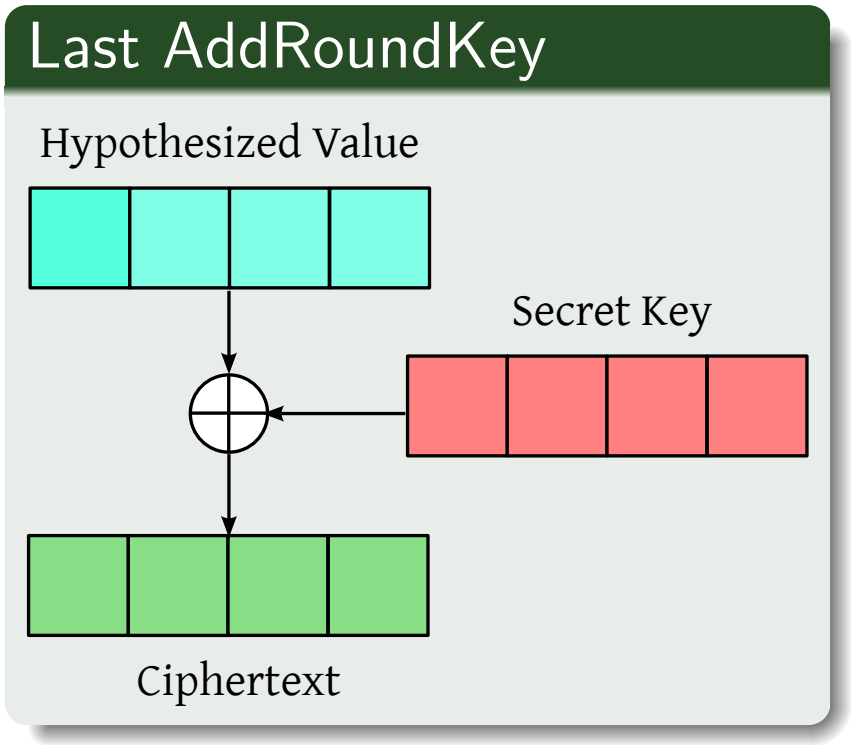
# Hypotheses on intermediate values

## Case Study: AES




We choose, as the value to be predicted, the input to the last AddRoundKey operation

# Hypotheses on intermediate values




- A convenient place to make prediction in AES-128 is the last AddRoundKey operation
- Since we know the ciphertext in full, we can predict some values of the state before the ARK, guessing the key bits
- Knowing one bit of the ciphertext, and guessing the value of a 8 key bits we will hypothesise a 8-bits of the cipher state

# Consumption modelling

- The dynamic power consumption scales linearly with the number of logic gates and the number of wires switching state
- Possible models for dynamic power consumption are:
  - Hamming weight of the predicted value: models the energy necessary to pull up the wires representing the result during the computation
  - Hamming distance between two different state bit values: models the switching activity of the D-latches (one bit registers) when they toggle to store the output of the operation. This model assumes that the previous value contained in the latches is known to the attacker.
- Both models fit reasonably well the dynamic power consumption of the circuit under the assumption that each bit takes the same amount of energy to be computed
- Notable exceptions (f.i. outputs of the DES S-Boxes) may show better correlation with a weighted hamming distance model

# Statistical hypothesis checking

- Once the attacker has gathered power measurements and computed power consumption hypotheses for a statistically significant number of inputs ($\geq 30$), he checks which one is the correct power consumption prediction
- Only the correct power consumption prediction will match the actual behaviour of the device, thus validating the guess on the secret key
- Hypothesis checking can be done through a variety of statistical tools. We will see **Pearson's linear correlation coefficient**

# Hypothesis checking - CPA

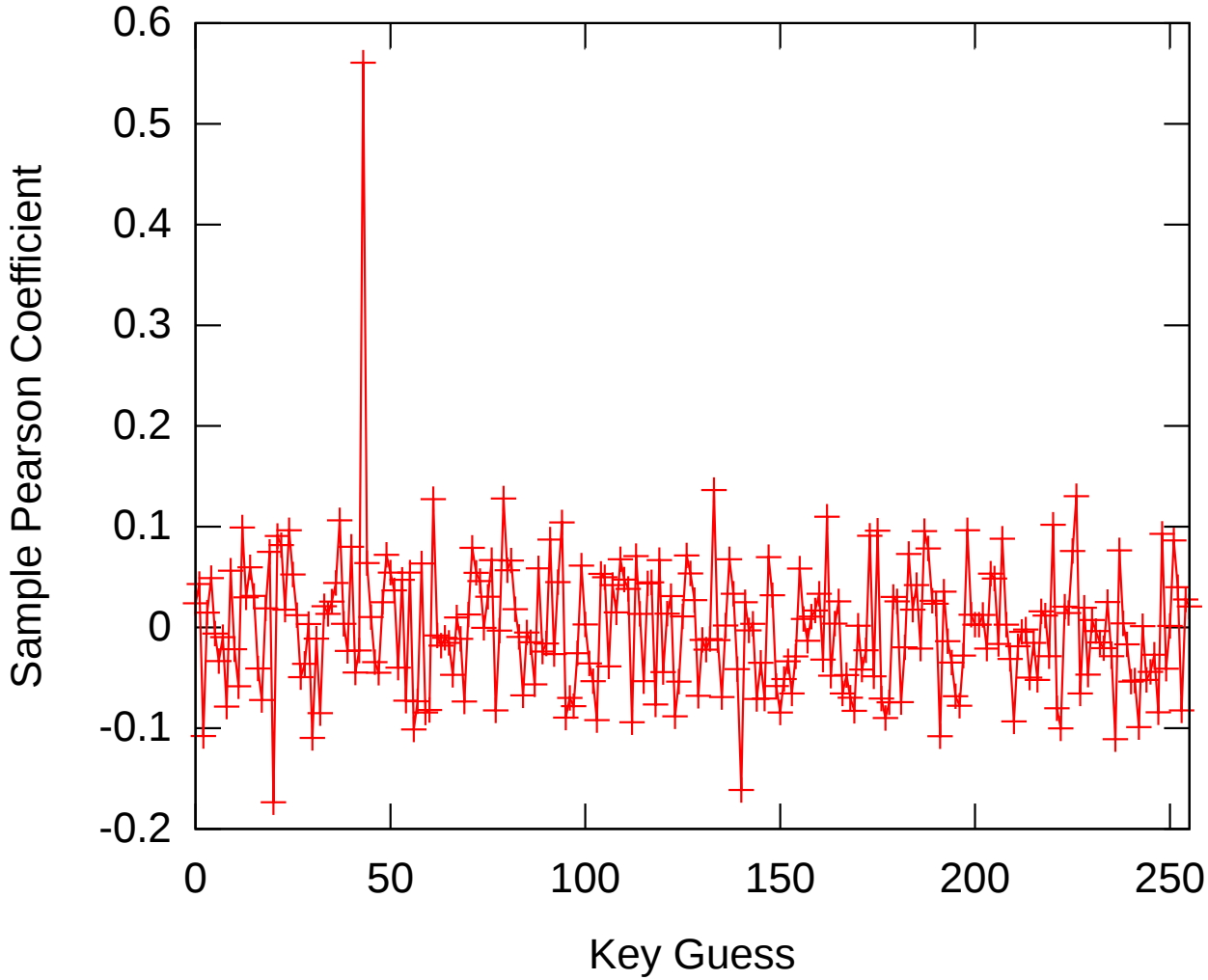



Figure: Sample Pearson correlation Coefficients for the attack against AES

# Relaxing Assumptions - Timing

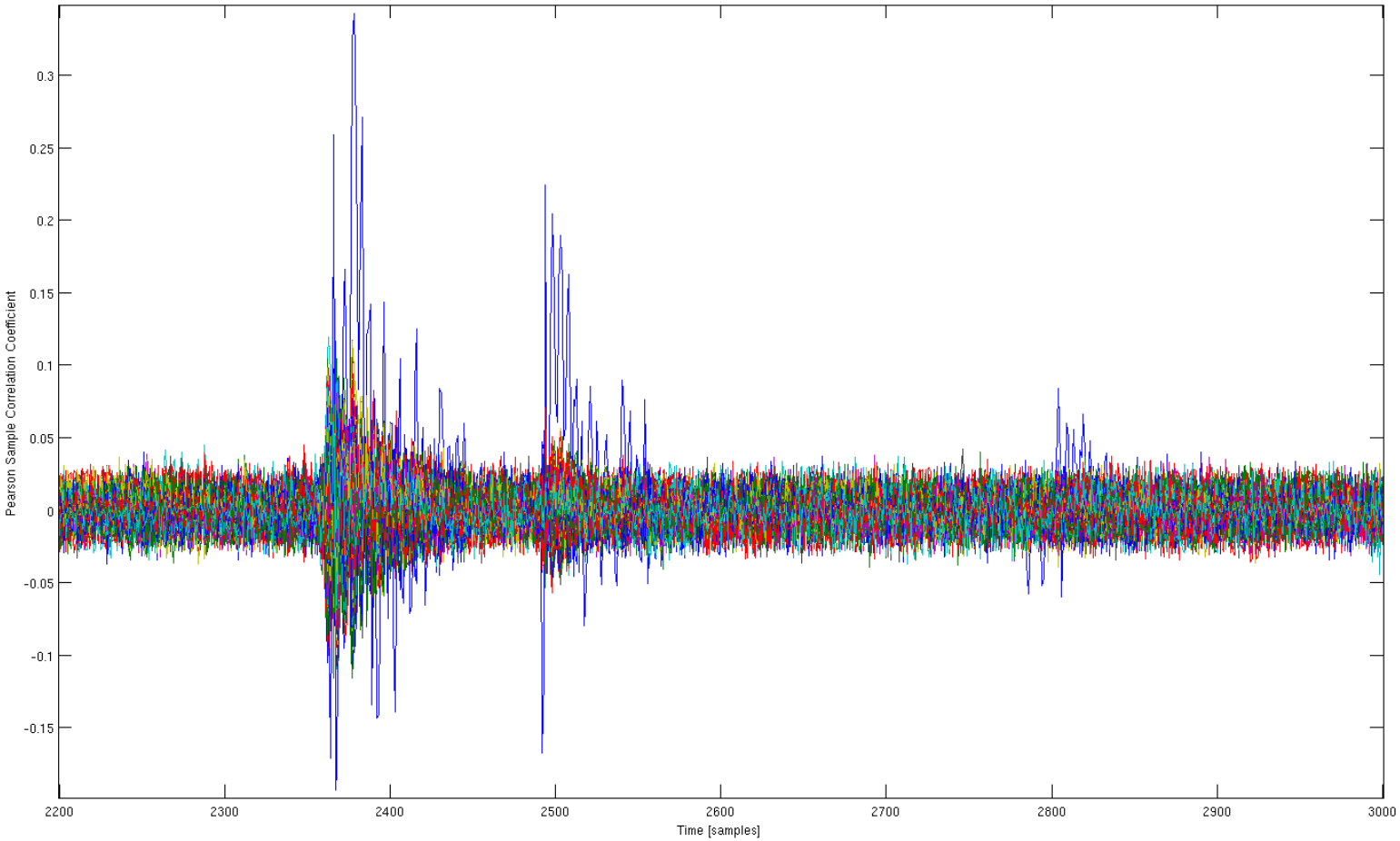



Figure: Sample Pearson coefficients for all the time instants sampled

# Automated Analysis

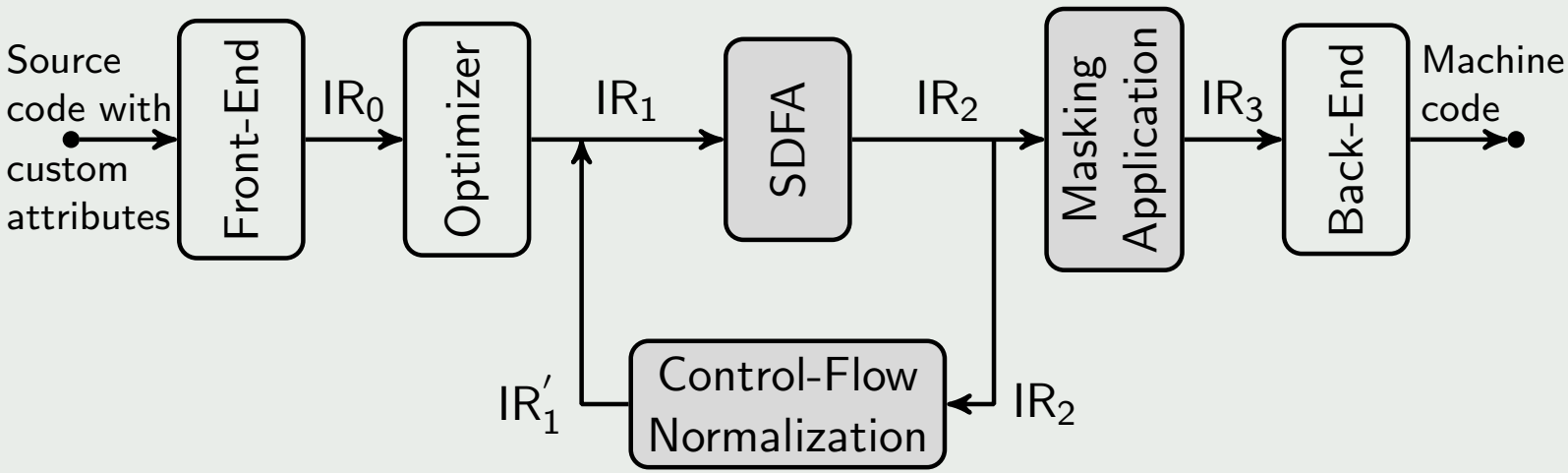
## Automating side channel vulnerability analysis

- Definition of a **Security-oriented data-flow analysis** (SDFA) assessing the *Instruction Resistance* of SW cipher instructions with bit-level accuracy
- Employ a LLVM pass to assess the instruction resistance *without profiling the power consumption* of the underlying platform
- **Automated application** of provably secure countermeasures only to the instructions needing protection

# New passes for LLVM

## Security-oriented Compiler Pipeline

- LLVM compiler pipeline with specialized passes to perform SDFA and countermeasure application
- Analysis and transformation performed on the LLVM IR



# Instruction Resistance

## Definition

Given a generic IR instruction $I$ with a $\mathrm{size}(I)$-bit output value

- The resistance $\mathcal{R}(I)$ is the **minimum number of key material bits** influencing any of the output bits of $I$
- The attacker needs to perform at least $2^{\mathcal{R}(I)}$ key guesses to predict a bit of the output of $I$
- If the output of $I$ does not depend on the key: $\mathcal{R}(I) = \infty$

## Example

```
int cipher(int k, int p) {
  int r=10;
  for (int i=0;i<r;++i) {
    p=p^k;
    p=(p<<8|p>>24);
  }
  return p;
}
```

## Key bits influencing the lsb

| $I$ | Round 1st | 2nd | 3rd | [4th, $r$-th] |
|---|---|---|---|---|
| xor | 0 | 0, 8 | 0, 8, 16 | 0, 8, 16, 24 |
| shl | 8 | 8, 16 | 8, 16, 24 | 0, 8, 16, 24 |
| ashr | | | | 0, 8, 16, 24 |
| or | 8 | 8, 16 | 8, 16, 24 | 0, 8, 16, 24 |

# Security oriented DFA

## Analysis step by step

1. Identify instructions defining key material
2. Identify instructions using attacker known material
3. Compute instruction resistance via forward SDFA
4. Compute instruction resistance via backward SDFA
5. Compute single instruction resistance

# Security oriented DFA

## Identify key material

- The first step is to detect which CFG nodes are defining key material (user key + values derived only from it)
- The variables containing the user key are marked in the source through a language attribute
- Clang lowers the attribute as metadata attached to the load operations of those variables
- The remaining key material nodes are computed adding to the key material node set $\mathcal{K}$ all the CFG nodes using only values defined by instructions in $\mathcal{K}$ until a fixpoint is reached

# Security oriented DFA

## Identify known material

- The second step is to detect the cipher nodes, i.e. the ones combining key material and inputs
- The plaintext is marked in the source code via a language attribute
- Clang lowers the attribute as metadata attached to the load operations of the plaintext
- The cipher node set $\mathcal{C}$ is initialized with the nodes marked with the metadata and all the CFG nodes which use at least a value defined by an instruction in $\mathcal{C}$ are added up to a fixpoint

# Forward DFA

## Local SDFA definition

- The forward DFA computes the dependence of each bit of the value defined by an operation from the key material
- The DFA is defined through the usual dataflow equations:

$$
\begin{aligned}
in(I) &= \begin{cases} \emptyset, & \text{if } pred(I)=\emptyset \\ out(J), & \text{if } pred(I)=\{J\} \end{cases} \\
out(I) &= \mathcal{F}_{op(I)}(in(I))
\end{aligned}
$$

  where $\mathcal{F}_{op(I)}()$ is a transfer function depending on the nature of `I`
- The dependencies of a single bit of the output of `I` from the key material are modeled as a vector of booleans

# Forward DFA

## $\mathcal{F}_{op(I)}()$ - bitwise operations

- For bitwise instructions combining two values, $\mathcal{F}_{op(I)}()$ adds the dependencies of the corresponding bits via inclusive or of their boolean vectors
- Bitwise instructions employing an operand are properly dealt with (e.g. masked out bits get their dependencies blanked)
- Load operations of key values simply initialize each bit of the loaded value as depending on itself
- Shifts and arithmetic shifts shift around the boolean vectors of dependencies along with the bits

# Forward DFA

## $\mathcal{F}_{op(I)}()$ - loads and arithmetics

- Load operations from lookup tables (common in block ciphers to compute nonlinear functions), where the index of the loaded value depends on the key diffuse the cumulative dependencies of all the input bits over all the outputs
- Additions and subtractions are considered, through a conservative estimate from a defender point of view, as bitwise operations
- Multiplications and divisions are considered to be diffusing the dependencies of their operand over all the bits of the result

# Forward DFA

## global definition and loop peeling

- To tackle a global SDFA a meet over all paths strategy is employed at control flow convergence points
- However, as this may cause a precision loss in the analysis, loop peeling is applied until the information coming from all the convergent edges is the same
- Conditional constructs, which are seldom appearing in block ciphers, are converted into straight instruction sequences through arithmetic-predication based if-conversion
- Note that in this case, the purpose of the if-conversion is to allow a deterministic analysis, and not to enhance performances, thus it is applied even if the if construct body has a significant size

# Backward DFA

## Same principle, reverse direction

- Since it is possible for an attacker to employ either the inputs or the outputs of a cipher to derive power consumption hypotheses, a Backward SDFA is necessary
- In this case, the starting point for an attacker are the store operations of the ciphertext
- The analysis is defined in a similar way to the Forward DFA, simply taking care of mirroring the effect of the dataflow equations
- loop peeling and if-conversions are applied in this case too (basically, the same normalized form employed for the forward SDFA is employed)

# Instruction Resistance Computation

## Results

- Once both the forward and backward SDFA are run, all the information on key dependencies is available
- If desired, the cipher implementer can examine the dependencies in detail
- To extract the instruction resistance $\mathcal{R}(\mathtt{I})$ the analysis
  - Counts the number of key bits on which an output bit depends according to both Forward SDFA and Backward SDFA
  - Selects the minimum amount of key bit dependencies among all the output bits of the operation as the resistance for $\mathtt{I}$
- Instruction with a resistance level below a sensible threshold (e.g. 80 bits) are the ones which should be protected

# Instruction Resistance Computation

## Results - AES