Politecnico di Milano
Scuola di Ingegneria Industriale e dell'Informazione
Corso di Laurea Magistrale in Ingegneria Informatica
Dipartimento di Elettronica, Informazione e Bioingegneria

Automated Side-channel Vulnerability Detection and Countermeasure Application via Compiler-based Techniques.

Relatore: Prof. Gerardo PELOSI
Correlatore: Ing. Alessandro BARENGHI

Tesi di Laurea di:
Massimo MAGGI Matr. 778986

Anno Accademico 2012–2013

Ringraziamenti

Questo lavoro è il risultato di un lungo periodo di costruttiva e intensa collaborazione con il Prof. Gerardo Pelosi e con l'Ing. Alessandro Barenghi, che mi hanno insegnato molto più di quanto sia ragionevolmente possibile trasmettere in un usuale corso da pochi crediti. Ma soprattutto mi hanno mostrato e fatto apprezzare i lati positivi e negativi del fare ricerca in un istituzione importante come il Politecnico di Milano. A loro va il più grande e sentito ringraziamento.
Ha inoltre avuto una grandissimo impatto positivo durante la mia permanenza in università l'aver fatto parte del *NECSTLab*, della squadra *Tower Of Hanoi* ed del *Politecnico Open unix Labs*. In queste occasioni ho trascorso momenti molto belli (tra cui gli interi weekend piacevolmente passati a giocare ai Capture The Flag, competizioni inter-universitarie di sicurezza informatica), e conosciuto molte persone, da entrambi i lati della cattedra, che hanno contribuito ad incrementare le mie competenze tecniche ed a rendermi una persona migliore. Tra i docenti meritano sicuramente di essere nominati il Prof. Stefano Zanero e l'Ing. Federico Maggi, che hanno creduto in me permettendomi di lavorare a progetti come il *PoliCTF* ed il *NECSTCloud*.
Viceversa non farò qui una lunga serie di nomi per quanto riguarda i colleghi, o più precisamente amici, che ho incontrato al Politecnico. Sono tantissimi ed il semplice inserimento dei loro nomi in una lista mi sembra assai riduttivo rispetto a ciò che meritano, ed a tutti loro va il mio ringraziamento.
L'ultimo, ma non per importanza, ringraziamento va ai miei genitori e ai miei amici frequentati al di fuori dell'ambito universitario per aver sopportato tutti i miei malumori e le mie angherìe nei momenti più intensi di questi cinque anni.

Sommario

L'obiettivo di questa tesi è di progettare e implementare una protezione automatizzata e ottimizzata dei cifrari a blocchi contro gli attacchi *side-channel*, eseguiti su sistemi *embedded*. Gli attacchi *side-channel* mirano a ottenere informazioni sul valore della chiave segreta raccogliendo dati sul consumo di potenza dei dispositivi di elaborazione, riconsocendo gli istanti in cui le istruzioni eseguite utilizzano valori della chiave crittografica. In particolare, la metodologia proposta analizza il *Data Flow Graph* (DFG) dell'algoritmo in esame con lo scopo di identificare come i valori intermedi calcolati dalle istruzioni dipendono da quelli della chiave segreta.
L'implementazione ha aggiunto nel compilatore LLVM un passo specializzato che lavora a livello di rappresentazione intermedia, permettendo l'analisi a prescindere dall'architettura su cui dovrà essere eseguito il codice. La validazione sperimentale ha consentito di applicare con successo una protezione selettiva ai cifrari a blocchi più comuni, e ha dimostrato un effettivo avanzamento dello stato dell'arte rispetto alle soluzioni correntemente adottate per rendere sicure le implementazioni software delle primitive crittografiche sui sistemi *embedded*.

Abstract

The target of this thesis is to design and implement an automated and optimized protection of block cipher implementations against side-channel attacks led on embedded devices. Side channel attacks try to obtain information about the encryption key through collecting power consumption (or EM emissions) data of the computing device. This target is reached through detecting when the executed instructions employ secret key material. In particular, the proposed methodology analyzes the Data Flow Graph (DFG) of the target algorithm with the aim to identify the dependencies of every instruction from the secret key material. The implemented toolchain added into the LLVM compiler a specialized pass that works at the intermediate representation level, enabling the analysis to be architecture agnostic. The experimental validation shows how to successfully apply a fine-grained protection to common block-cipher suites and validated an effective advancement of the state-of-the-art with respect to current solutions adopted to secure the software implementations of cryptographic primitives on embedded systems.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Few people remember a great advantage that led the Allies to the victory against the Axis powers in World War II: the decryption of Enigma-encrypted messages, thanks to Alan Turing.
Since those days, cryptography acquired an increasing importance, not only on military environments. Mobile phones using the old GSM standard encrypted all the communications, and had a small secure token to identify users: the SIM card.
This card is not a simple memory containing phone number and phonebook. It is a fully featured CPU which encrypts data and is designed never to reveal its own private key.
Sadly, SIM cards have a history of being cloned by malicious individuals. There are a lot of examples of other devices using similar technologies: credit cards, tickets for public transportation, Satellite TV cards and even contactless micropayment systems (e.g. coffee machine's tokens). Their proven vulnerability is one of the reasons why we should work to improve security on those devices.
As it has been recently disclosed to public opinion, the privacy of people all over the world today is significantly compromised by a government agency. We can assume that they are doing it exclusively for national security purposes, but how can we be sure that their will retain an ethical behavior forever? What defenses do we have? Strong cryptography, which implementations are publicly audited regularly by experienced and knowledgeable people. In other words, strong reliance on open source software.

Modern compilers are extremely large and complex software. They are the only category of software that can create a better copy of themselves. Think about this situation: a programmer adds a great optimization to a compiler. He compiles the compiler, then the new compiler is employed to obtain a new copy of itself, more performing due to the optimization. Compilers have already started to autonomously learn how to improve themselves [26].
Compilers usually do more than producing assembly code from the source code: they check the code for syntax errors, undesired programming practices (warnings), they can optimize programs for different objectives: typically performance or small code size, but another goal could also be energy savings [2].
They also have been proven successful in improving application security: many security mechanisms have a consistent part of the work done by compilers, for instance the compiler pass (in GCC and LLVM) called AddressSanitizer [24] which checks for correctness every single memory access done by a program, disallowing a wide range of exploits. Most modern compilers can insert "canaries" or "stack guards" to prevent exploitation of buffer overflows in the stack. An example of stack protector is the *ProPolice* [11] pass that is part of GCC since version 4.1, released in 2007.
The compiler infrastructure can be also used for developing code completion engines and code refactoring engines, which greatly improve programmer's productivity.
In this work, we are going to use the compiler infrastructure of the *LLVM Framework* [16] to analyze the implementations of block ciphers from the point of view of side channel leakage, and fix them as efficiently as possible to run the primitives on embedded devices. In practice, we are adding another optimization goal: security. It hinders performance and code size, but sometimes it can be more important than them. Let's give system designers more choice.
What are side-channels? Side channels are means to (voluntarily or not) send auxiliary informations about the information given through the *main* channel.
When you are talking, your blood pressure, pulse, breathing frequency, and skin conductivity varies greatly during the conversation, mainly in relation to your emotions. You are giving out an information that you are trying to

hide: if what you're consciously saying is a lie or not. This is the principle behind the polygraph, also called lie detector.
When CPUs, especially small ones, are encrypting, their power consumption, electromagnetic emissions, response time and reactions to purposely induced faults significantly varies during the computation of the cryptographic primitive, mainly in relation to the value of the secret key. In other words,the CPU is giving out information that it is trying to hide in the computed ciphertext: i.e. the encryption key. This is the effect that is going to going to be studied and mitigated with this work.
This is a fairly recent research topic, as first articles on it were published in late 90's. As of today, the existing countermeasures are partially effective and their use significantly impacts performances and code size. In this work, a new methodology for the assessment of the power-based side-channel leakage has been developed. The developed tools and the extensive experimental campaign put into effect to demonstrate the effectiveness of the approach, and enabled to obtain substantial improvements in the engineering of software-based cryptosystems. The main results have been published in one major international conference on the topic. [1]

Chapter 2

Technical Background and State of the Art

This chapter will provide some useful background notions to understand the rest of the work. This also describes the current state of the art, as the exploration of side channels is a rather recent topic. The background notions include some details on the inner structure of compilers and the algorithms and data structures they employ to optimize programs. Moreover the background on block ciphers as well as a summary of the usual workflow employed to led a side-channel attack, is provided. Finally, to build efficient countermeasures, we should also know what "masking" is and how it is actually applied.

2.1 Compilers

Definition 2.1.1 (Compiler). A compiler is a program that translates the source code written in a high level language into a target language, usually an assembly language tailored to a specific platform.

The compiler is usually organized in three stages: front-end, middle-end (Optimizer) and back-end, as pictured in Figure 2.1 The front-end recognizes tokens in the source code, performs syntax checking and builds the Abstract Syntax Tree.
The Abstract Syntax Tree is a tree representation of the entire program,

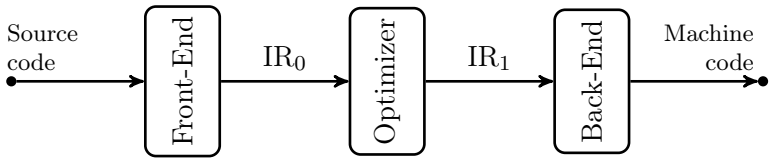


Figure 2.1: Overview of a compiler structure

which is then visited to emit the unoptimized Intermediate Representation of the program.
The Intermediate Representation (IR) is usually a programming language similar to assembly but not tied to any specific hardware. Therefore it has a slightly higher level of abstraction, which allows to optimize code in an efficient manner. This facilitates the re-use of the optimizer for different source code languages.
The IR is then optimized in various steps and then translated (by the back-end) to the target assembly language. Finally, the assembly source code is assembled into an object file which in turn is linked together into a single executable.

2.1.1 Data Structures

The most frequently used data structures, needed in the following chapters, are the Control Flow Graph and the Data Flow Graph. These data structures, which may look similar at a first glance, have an extremely different meaning: The first represents the order in which instructions are executed, and the second represents the data dependencies among them.

Control Flow Graph

The Control Flow Graph is a representation, of the control flow dependencies among the instructions of a subprogram. Each node in the graph represents an instruction, and each edge outgoing from a node represents a possible path to be followed diversing the execution.
Those instructions can still be reordered by a subsequent scheduling step, prior to be linked into one executable. However for instructions with multiple outgoing nodes (f.i. conditional branches) the next instruction that will be executed is determined at runtime, and possibly based on the evaluation

of input-data dependent conditions.
At each execution of an instruction with multiple outgoing edges, it's possible that the chosen outgoing edge is different.
The instructions represented in the CFG may be machine-level instructions, Intermediate Representation instructions or high-level programming language statements, depending on the designer needs.
Usually, CFGs are related either to machine-level instructions or IR instructions.
In this thesis we'll assume that everything is related to IR instructions.
Figure 2.2 shows how the reported CFG is linear up to the BGT (Branch if Greater Than) instruction, which can jump to two different basic blocks. In this example, after the end of one possible basic block (i.e. the one with the ADD %R3,10 instruction) the control flow returns to the same basic block where the other possible choice jumps to.

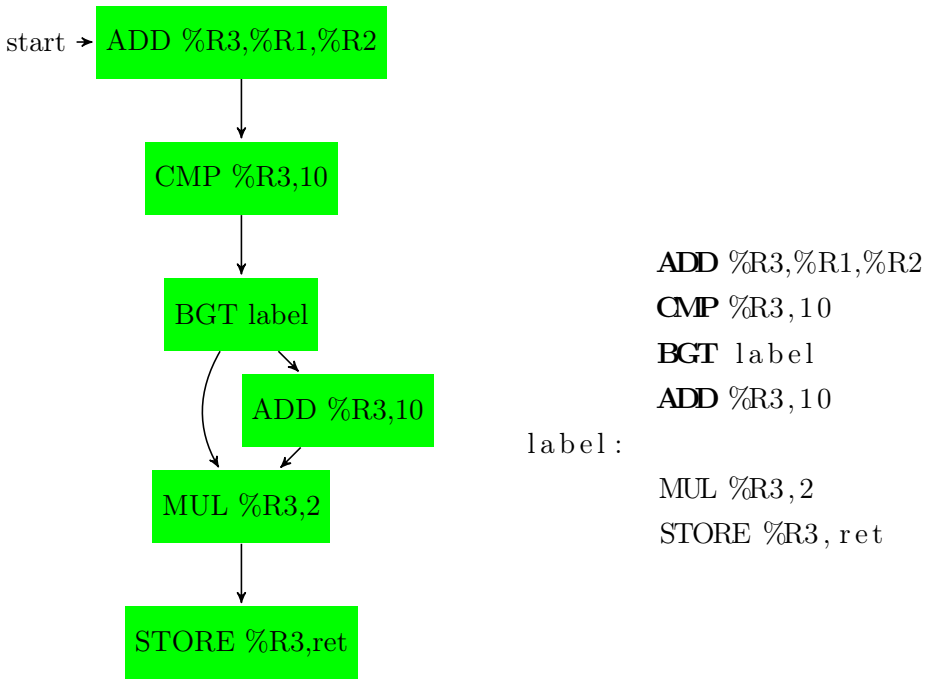


Figure 2.2: Sample CFG with corresponding code snippet. The instruction executed after *BGT label* can be an addition or a multiplication, based on the content of the previous compare. Also, after the (eventual) addition, the final multiplication gets executed anyways.

Data Flow Graph

Definition 2.1.2. A Data Flow Graph (DFG) is a graph in which each node represents an instruction, while the edges represent data dependencies between the involved instructions.

Definition 2.1.3. A data dependency is the relation between the instruction that takes as input a certain value and the instruction that defines (i.e. computes) it.

A data dependency between two instructions means that the instruction that defines the value should be executed before the one that uses it, but not strictly before. The effective sequence of instructions in the executable is determined by the scheduling step, considering both the Control Flow Graph and the Data Flow Graph.

Dataflow Analysis

Definition 2.1.4. A Dataflow Analysis takes as input the DFG and derives a property of the data dependencies of a program to understand how it manipulates its input data.

Examples of the common DFAs are the following:

- Reaching definitions analysis: Its purpose is to detect which definitions of a variable can reach a particular use. This is repeated for every use of every variable. In simpler words, which instructions could have defined the values that I'm going to use?
- Constant propagation analysis: It determines if the uses of a variable (that by definition needs to be read from memory) can be replaced with a constant. It re-uses the reaching definition analysis.
- Live variables analysis: It determines if a certain variable (or intermediate value) is needed in the following computations. This should not be confused with Garbage Collection, which is a completely different concept. GC works at runtime, while live variable analysis works at compile time.

- Available expressions analysis: It determines which expressions were already evaluated and establish if their value is already available. It is the helper analysis for the optimization pass denoted as: *Common Subexpression Elimination*, which tries to not compute two times the same subexpression (in case it is pure, i.e. without side-effects)

Static Single Assignment form

Definition 2.1.5. The Static Single Assignment (SSA) form is a normal form for Intermediate Representation languages, in which each virtual register is written exactly once.

Each instruction produces exactly one result, and each register (or value) is written exactly by one instruction, so we can assume a 1-to-1 relation between instructions and their output registers.
Thanks to this relation, the concepts of *instruction* and *register*, despite being substantially different, can be thought of as a single entity.
Note that, some instructions do not produce data by design, e.g. `jump`,`return` or `store` instructions.
It is possible to introduce two definitions:

Definition 2.1.6. The definition of a SSA register represents the value returned by the instruction generating it. It represents the unique write of a value into a specified SSA register.

Definition 2.1.7. Uses of SSA registers are the set of read operations done on the register. They appear as input arguments of other instructions.

Definition 2.1.8. A basic block is a list of instructions executed strictly in sequential order.

This 1-to-1 relation fits perfectly for managing a linear CFG, but shows its limits with programs having a non-linear CFG. For instance, consider the following high-level code:

```
A=3*3;
if (B>2){
  A=2*2;
```

```
  }
C=B+A; //Can you guess what is the correct
       //SSA form?
```

This problem can be solved introducing $\phi - nodes$, that are abstract instructions which return a different value depending on the previously executed basic block. $\phi - nodes$ take as parameters a list of pairs (value,basic block). Using this abstraction it's possible to represent all of the usual language constructs (if, for, $while$, etc..) by placing a $\phi - node$ at the start of each basic block, for each used value that can be defined by two or more different basic blocks.
This form is simpler to manage when optimizing code, and makes many data flow analysis (f.i. reaching definitions and liveness analysis) quite simple and efficient.
The SSA form assumes that the machine has an unbounded number of registers. This is not a problem, as it it applied at an IR level and not on assembly-level language. The back-end of the compiler, which translates the IR into the actual assembly code has a pass called *Register allocation* which determines which SSA registers should be assigned to physical registers, and manages the insertion of `load` and `store` actions to fetch and save the correct values from the stack, in case the physical registers cannot be reused to satisfy the data dependency constraints.

2.1.2 Low Level Virtual Machine

This subsection details the inner structure of the LLVM infrastructure, which is used extensively in this work. The LLVM project [16] started in 2000 at University of Illinois as a research project focused on static and dynamic compilation of a variety of languages. Being highly modular, it was used a lot in industry and academic fields.
In 2005, Apple hired Chris Lattner, one of the designers of LLVM. He was hired to bring LLVM to production quality for Apple needs. LLVM made its appearance in mainstream Apple products in 2007, with Mac OS X 10.5 where it powered the JIT compiler of the OpenGL stack. It started to appear in the toolchain shipped with Xcode starting with version 3.1.
In May 2012 LLVM/Clang 3.1 were released, including AddressSanitizer [24]

developed by Google. AddressSanitizer is a memory error detector, which is typically used to find security vulnerabilities in compiled programs in an automated way. It's average slowdown compared to a non-instrumented executable is 2×, which is an order of magnitude faster than competing solutions.
Since 6 November 2012, Clang is the default front-end compiler for FreeBSD on x86 systems, both for kernel and userspace programs. Also, the NVIDIA CUDA Compiler is based on LLVM, and thanks to modularity of LLVM, developers can easily create front-ends for new programming languages targeting NVIDIA's GPUs.

In the LLVM Framework the compilation is a three-stage procedure:

- Front-end: Language-specific translation of the source programming language to LLVM IR. The most known front-end to LLVM is *clang*, which can compile C, C++ and some of their dialects.
- Optimization: This step is language-independent. The unoptimized LLVM IR produced by the front-end gets optimized by running it through a set of passes.
- Back-end: The now-optimized LLVM IR gets translated to the assembly of a specific target.

Most of the work presented in this text takes place in the optimization stage. It is organized as a sequential run of different passes, with dependency management provided by the framework.
The set of passes to run is usually decided by the user: for instance the optimization levels `-O1, -O2, -O3, -Os` are predefined set of passes to run. The passes can be divided into two categories:

- Analysis passes: They provide informations about the code to be optimized, without modifying it.
- Optimization passes: They modify the code, using informations provided the by analysis passes.

Each pass should declare its own dependencies and which analysis they preserve in order to schedule passes in the right order. Preserving an analysis

means that the pass can change the code but asserts that the information provided by a specific analysis is still valid for the now-modified code.
The LLVM IR is in Static Single Assignment form (see paragraph 2.1.1).
It exists in three different forms, with exactly the same expressivity: textual, binary (bitcode) and in-memory representation.
The LLVM IR is a strongly typed language, which means that every data being manipulated has an associated type.
Primitive types are numerical ones (integer of different sizes and floating point) and they can be composed in arrays, structures or pointers.
In this language is possible to attach (one or more) metadata nodes to each instruction. Those nodes may have a name and usually contain one or more typed values (strings are allowed).
Metadata is preserved across all three representations of IR and is used for "Type Based Alias Analysis", loop description or variable value constraints.
A developer can create new metadata nodes to represent customized informations.
Here you can find a quick reference of most frequently used LLVM IR instructions:

- Basic block terminator instructions
  - `ret`: return from the current function. Its argument is employed as the return value.
  - `br`: Either an unconditional branch (with one argument and a single destination label) or a conditional one. The conditional form takes three arguments, an $i1$ typed value (integer of size 1, semantically equivalent to a boolean) and two labels.
- Binary Operations (the f prefix is intended for floating point values).
  - `{f,}add`: Takes two arguments and returns their sum.
  - `{f,}sub`: Takes two arguments and returns their difference.
  - `{f,}mul`: Takes two arguments and returns their product.
  - `[usf]div`: Unsigned or Signed division operator.
- Bitwise Binary Operations

  - `shl`: Shift left.
  - `lshr`: Logical shift right.
  - `ashr`: Arithmetic shift right: Operates a correct sign extension when shifting signed values
  - `and`, `or` and `xor`: Well-known bitwise binary operators.

- Memory Access and Addressing Operations
  - `alloca`: Reserves the requested amount of space in the stack for a local variable.
  - `load`: Reads from memory. Takes a pointer as a parameter, and the size is implicitly given by the size of the type. Returns the read value.
  - `store`: Writes to memory. Takes a pointer and a value of the corresponding type as parameters. Does not return a value.
  - `getelementptr`: Operates pointer arithmetic using type informations to obtain pointers to particular elements in arrays or in structures, even if deeply nested.

- Conversion Operations
  - `trunc .. to`: Returns the value passed as first parameter truncated to the (smaller) type specified as second parameter.
  - `[sz]ext .. to`: Returns the value passed as first parameter extended (with Zero or proper Sign) to the (larger) type specified as second parameter.
  - `bitcast .. to`: Returns the value (or pointer) passed as first parameter but with type specified in the second parameter. The type should have the same size.

- Other Operations
  - `{f,i}cmp`: compares two Integers or Floating point numbers and returns an $i1$ with the boolean result of the comparison.
  - `phi`: creates a $\phi - node$ (see paragraph 2.1.1)

- `select`: Takes three parameters: an $i1$ and two other values. Return one of the two values depending on the boolean value.Frequently used after `{f,i}cmp`.
- `call`: Calls a function and returns it return value.

LLVM also includes some special functions (to be invoked with the `call` instruction) denoted as *intrinsics*. These intrinsics represent operations that might be done in a particularly efficient way in some processors, while requiring library functions in others.
Some intrinsics are mathematics functions (`sqrt`, `sin`, `cos`, `pow`, `exp`, `log10`, `log2`, `fma`) or bitwise operations (calculate Hamming weight, count leading/-trailing zeroes).
They can also be used to mark breakpoints to request the debugger attention or mark the lifetime of memory objects to help a garbage collector to perform its task.
The translation to native code of those intrinsic functions is highly target-dependent.

Code organization

LLVM is developed in C++, using a mostly-flat namespace. It uses a customized form of RunTime Type Information (RTTI), for performance reasons. So usual C++ casting operators are unavailable, but equivalent ones (`isa<>`, `cast<>`, `dyn_cast<>`) are provided. It uses C++ STL but also provides some customized containers. Usually each class provides either standard iterators for each of its members or logically linked items, like:

- `llvm::Module::begin()` returns an iterator for functions contained in the module.
- `llvm::Function::begin()` returns an iterator for basic blocks.
- `llvm::BasicBlock::begin()` returns an iterator for instructions.
- `llvm::User::op_begin()` returns an iterator for operands of an instruction. Note that `User` is a superclass of `Instruction`, so this iterator is present on them as well.

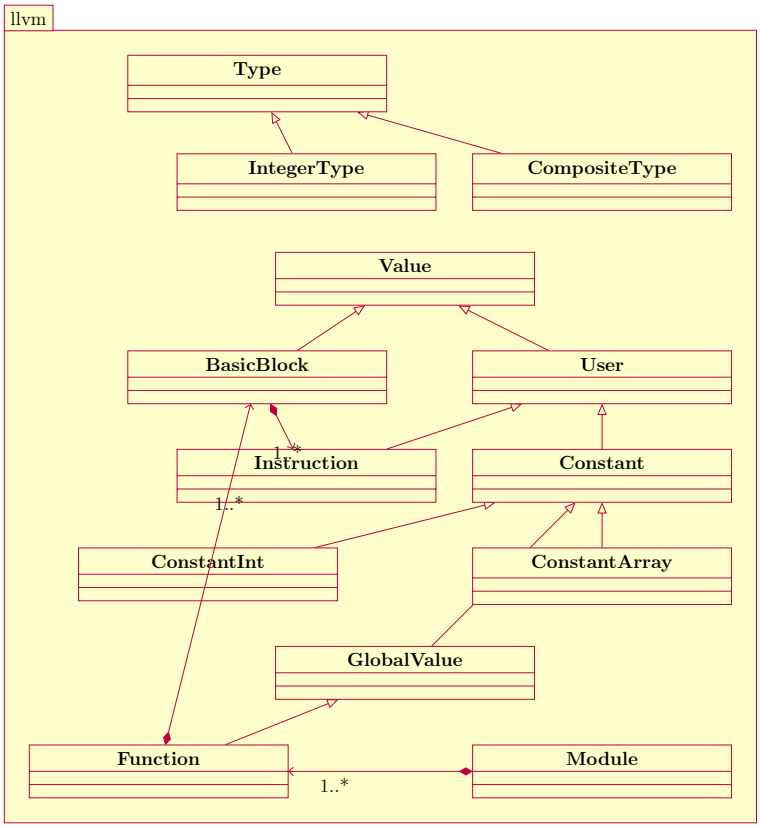


Figure 2.3: UML Representation of some frequently-used classes in LLVM

- `llvm::Value::use_begin()` returns an iterator for all known uses of a value. Note that `Value` is a superclass of `User`, which is a superclass of `Instruction`, so this iterator is present on them as well.

2.2 Cryptographic Primitives

This section will contain a detailed description of block ciphers, that will be the subject of this study. Cryptography is a really large science, which includes many other topics, like asymmetric ciphers, stream ciphers and hash functions. Asymmetric ciphers use two different keys, a public one and a private one. They allow everyone to encrypt data which can be deciphered only by the intended recipient and allow cryptographically-strong digital signatures. Those algorithms are usually based on arithmetic operations, and have strong control flow dependencies, which makes them ideal targets for Simple Power Analysis (SPA). Stream ciphers are symmetric ciphers exactly like block ciphers, but they solve the problem in a different way: they produce a random stream that depends only on the key. In an ideal scenario, from this stream it should not be possible to recover the key. This stream is then $\oplus$-ed (XOR-ed) with the data that needs to be encrypted, on a principle similar to One Time Pad. Hash functions are structurally similar to block ciphers, but they allow to obtain a fixed-size tamper-proof checksum of data. Some hash algorithms are used as "keyed hashes", in which a private value is involved. They allow to *authenticate* a chunk of data, using a key that should be shared between the *prover* and the *verifier*, unlike the much slower asymmetric algorithms. Due to the structural similarity, in future this work could be extended to analyze also hash function implementations.

2.2.1 Block Ciphers

Block ciphers are algorithms that process a fixed-size quantity of bits (a block) with a reversible transformation characterized by a parameter called: cipher key.
This transformation should be reversible only by knowing the same key that was used to encrypt the data, provided that the algorithm is correctly designed. The key management in a system where only block ciphers are analyzed, is simpler however the recipient needs to receive the key though a

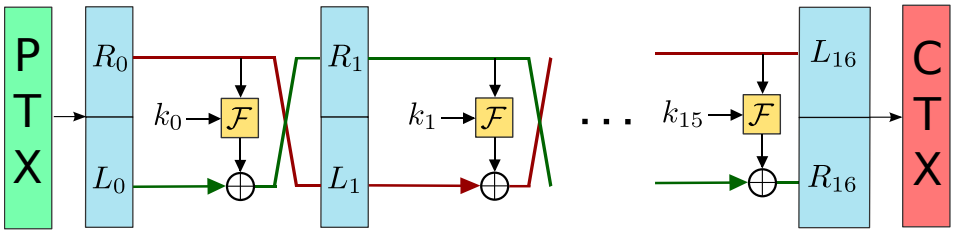


Figure 2.4: Block diagram of a Feistel cipher.

pre-existing secure channel.
The main advantage of block ciphers against asymmetric ciphers is the speed, as they are roughly 3 orders of magnitude faster. When the user encrypts data with a program that uses asymmetric keys, like GnuPG, it usually encrypts the user data with a symmetric block cipher, using a randomly-generated session key. This (much smaller) session key is then encrypted with the asymmetric algorithm requested by the user.
Most block ciphers design structures fall into two categories:

- Substitution Permutation Network (SPN)
- Feistel network

The Substitution Permutation Network represents a cipher, composed by multiple identical passes called rounds, where each one has three mandatory components:

- S-Box: Substitution boxes, which are nonlinear functions, whose effect is to introduce confusion into the cipher, so allow one-bit change in input to change a lot of bits in its output.
- P-Box: Permutation boxes, introduce diffusion into the cipher, scrambling the order of the bits into the block.
- Key addition: adds key dependency between each round. Usually it's a $\oplus$.;

The *Feistel network* structure, represented in Figure 2.4, is known since 1973. The designer of the algorithm has to define only the F function, usually identical for each round.
A Feistel network starts by splitting the input in two halves, applying the

customized function to one half and bitwise adding via xor the output with the other half. Then the two halves are switched and the next (identical) round starts.
The main benefits of a Feistel network are:

- Simple decryption algorithm: It's enough to invert the key schedule and swap the two halves of the ciphertext block before feeding in into the same cipher.
- The F-function does not need to be invertible.

Some of the most known Feistel-based ciphers are:

- DES and its derivatives (3DES, DES-X) (National Security Agency preferred block cipher from 1979 to 1999, and standardized by NIST)
- GOST 28147-89 (Russian government standard block cipher)
- Camellia (Japanese government standard block cipher)
- Blowfish/Twofish
- CAST family.

Advanced Encryption Standard

In 1997 the National Institute of Standards and Technology of the United States (NIST) realized that the DES, with its 56-bit key, was becoming practically vulnerable to bruteforce attacks.
They started an open process to choose "an unclassified, publicly disclosed encryption algorithm capable of protecting sensitive government information well into the next century"[1] The technical requirements were:

- It has to be a symmetric block cipher
- Its key length can be extended in an easy way
- It should be easy to implement in both hardware and software.

[1]Source: http://csrc.nist.gov/archive/aes/pre-round1/aes_9701.txt

There were 15 proposed algorithms from all over the world.
On 2 October 2000 the NIST announced that the winning algorithm was Rijndael, developed by Belgian cryptographers Vincent Rijmen and Joan Daemen.
The algorithm itself is a Substitution-Permutation-Network, works on 128-bit sized blocks and is standardized for key lengths 128,192 and 256.
It has a different number of rounds depending on the key length:

- 10 rounds for 128 bit key length
- 12 rounds for 192 bit key length
- 14 rounds for 256 bit key length

Its key scheduler sets the user key as the first bits of the subkey material and iteratively produces new subkey material until the needed quantity is reached using the previous subkey material as input. The subkey chunk size in the key scheduler is different from the subkey size used in the core algorithm itself (fixed to 128 bits).
The new subkey material is created from the previous one using combinations of rotations, XORs, Substitution boxes and exponentiations in a finite field.
The initial content of the state of the algorithm (represented as a 4x4 matrix of bytes) is the plaintext, while the final content is the ciphertext.
The algorithm is composed of a variable number of round, and in each one:

- SubBytes: Each individual byte is replaced with the 8-bit non-linear SBox. The Rijndael SBox (differently from other encryption algorithms,such as DES) has a precise mathematical structure, and is defined as the multiplicative inverse of the 8-bit value over $\mathbb{F}_{2^8}$ using the polynomial $x^8 + x^4 + x^3 + x + 1$.
  In most implementations the SBox is stored as a table to improve performance. The non-linear mixing provided by the SBOX provides the confusion effect as defined by Shannon [25].

- ShiftRows: The cells are shifted in the same row by a quantity equal to the index of the row. This is important to have linearly independent columns.

- MixColumns: It provides a $\oplus$-linear intra-column diffusion: each column of the state is multiplied by a constant matrix (with the rules of a proper algebraic structure) Every column I is considered as coefficients of a polynomial over the ring $(\mathbb{F}_{2^8}[X], +, \cdot)$,
  i.e : $I(X) = i_0X^0 + i_1X^1 + i_2X^2 + i_3X^3 \mod (X^4 + 1)$. Every coefficient i_i lies on $(\mathbb{F}_{2^8}, \oplus, \odot)$, and is expressed as a byte. I is multiplied by a fixed polynomial $C(X) = 0x02X^0 + 0x01X^1 + 0x01X^2 + 0x03X^3$.

- AddRoundKey: The current subkey (size fixed to 128 bit, equal to the state size) is combined bitwise using XOR with the state.
  This step adds dependencies from the key.

There is also a *round 0* that consists only of the AddRoundKey, while the last round skips the MixColumns step.

Serpent

The Serpent block cipher [3] works on four 32 bit words (for a total block size of 128 bits) and is based on a substitution-permutation network employing 32 rounds plus an initial and final permutation. It was a finalist of the Advanced Encryption Standard contest, where the current AES was preferred for its relative computation speed on small blocks. The round function of Serpent is constituted of a nonlinear step, in the form of 4-to-4 bit tabulated nonlinear functions, and a linear diffusion layer realized with bitwise rotations, shifts, and xor operations. Algorithm A.0.1 shows the structure of the algorithm.

Camellia

The Camellia algorithm (see A.0.4 at page 102)is a 128-bit block cipher developed by Japanese researchers of Mitsubishi and NTT. The algorithm, in case of a 128-bit secret key, employs a mixed round strategy, encompassing 18 Feistel rounds, a nonlinear bitwise transformation, denominated FL transformation in the standard and its inverse. More in detail, the algorithm performs a pre-whitening of the cipher state through adding via xor a portion of key material, subsequently it computes 6 Feistel rounds on the state. Following those, the cipher state is subject to the FL transformation, and

the output is processed via 6 further Feistel rounds. Finally, the output of the second batch of Feistel rounds is mapped through an inverse FL transformation and further processed by 6 Feistel rounds and an output whitening via bitwise xor.

DES, 3DES and DES-X

The Data Encryption Standard (`DES`) cipher is a symmetric block cipher with a 64-bit block and a 56-bit key [20]. It was chosen as US federal standard by NIST in 1977, when a key space of 2^{56} items was considered to be a good choice to render any brute-force attack unfeasible. The `DES` encryption/decryption algorithm is an iterated block cipher consisting of 16 rounds, each designed with a Feistel structure, which processes the left half of the block with through a nonlinear function, the so-called Feistel function, and combines the result via `xor` with the right half of the block. This result is employed as the right half of the input to the next round, while the old value of the right half is used as the left half of the block input to the next round [20]. The Feistel function, performs an initial expansion on the 32-bit value obtained in input, resulting in a 48 bit value, to which 48 bits of the secret key are added via `xor`. The result is split into 8 portions, each of them 6 bit wide, to which a 6-to-4 bit nonlinear function is applied. The 48-bit round keys are obtained via a key schedule algorithm which at first permutes the key bits (Permuted Choice 1, PC-1), discards the eight parity bits and divides the key material into two 28-bit halves; each half is thereafter treated separately. In successive rounds, both halves are rotated left by one or two bits (depending on the round), and then 48 *subkey* bits are selected through a second fixed permutation (Permuted Choice 2, PC-2). A different set of key bits is used in each subkey (one for each round of the encryption/description algorithm) in such a way that each bit is used in 14 out of the 16 subkeys.

The cipher starts processing the 64-bit plaintext via an initial bitwise permutation (`IP`) of its bits, the output of which is divided into two 32-bit blocks in order to serve as input of the first round. Following this 16 rounds are computed as described before, and the left and right halves of the results are swapped after the end of the computation. The result is thus subject to a final bitwise permutation ($\texttt{IP}^{-1}$) to generate the 64-bit ciphertext. A peculiar property of the `DES` cipher is that the decryption algorithm is identical to the

encryption one, except for the fact that the round keys should be employed in reverse order.

From a software execution point of view, the `DES` block cipher is characterized by a large number of bitwise operations, which have intentionally been designed to be hard to execute on software platforms.

DES-X and Triple DES are both derivatives of DES designed with the aim of increasing the key length over the original 56 bits of DES, while taking advantage of the existing DES implementations (in particular, of the existing hardware implementations, as both DES-X and Triple DES perform only simple operations beside the DES). Triple DES applies DES three times (the second time using the decryption round key order) using three 56-bit keys K_1, K_2 and K_3. When $K_1 = K_2 = K_3$ (keying option 3) Triple DES behaves exactly as DES, otherwise the effective key size is increased to 112 or 168 bits, depending on whether $K_1 = K_3$ (keying option 2) or not (keying option 1). DES-X augments DES with two additional 64-bit keys K_1 and K_2, which are xor-ed to the DES plaintext and ciphertext respectively to obtain the DES-X ciphertext. Note that the effective security margin provided by DES-X against brute force attacks against is actually only marginally better than Triple DES under keying option 2 as a meet-in-the-middle attack is particularly efficient on it.

GOST 28147-89

The GOST 28147-89 standard [27] defines a 32-round Feistel network block cipher used in the Russian Federation and in other countries of the former Soviet Union. The cipher, depicted in Figure A.0.3 at page 102 processes a 64 bit plaintext block employing a 256 bit fixed length key, which is mixed, in 32-bit-wide portions, adding one of them modulo 2^{32} as the first operation of the the Feistel function. Subsequently the 32-bit half of the state is passed through eight 4-to-4 bit S-boxes, and left rotated bitwise by 11 bits.

CAST5

CAST5 is a block cipher used by the Canadian government and is the default choice of symmetric cipher in the GnuPG cipher suite. Algorithm A.0.2

shows the structure of the cipher, which is a Feistel network with 16 rounds using a 64-bit block size and up to 128-bit key size. The algorithm works on 32-bit words, and employs arithmetic operations in addition to substitution maps and bitwise rotate operations in the Feistel network.

2.3 Side Channel Attacks

This section will contain a description of side-channel vulnerabilities, i.e. techniques to get (part of) the key used to encrypt or decrypt data using informations not present in the ciphertext. Most of those attacks must be conducted in physical proximity of a device doing encryption operations, however they should not be underestimated. Did you know that the electrical power company can know which movie are you seeing at home? [12]
Attacking a cipher through via side-channel attacks means recovering a concealed information (usually the key) using an information source other than the ciphertext. Typically those informations are gathered while the block is being encrypted through:

- Power consumption of the CPU
- EM emissions from the CPU
- Timing anomalies
- Optical emission from the die of the CPU
- Results from purposely faulty encryptions.

The side channel attack techniques are traditionally split in two categories: Passive (carried through *observation* without interfering with the chip functionality) or active (inducing faults).

2.3.1 Passive Attacks

Common passive side channel attacks are:

- SPA (Simple Power Analysis): The key is extracted by exploiting variations in the power consumption caused by key-dependent **control flow**. Most block ciphers don't have a key-dependent control flow, so

this attack is mostly useful against asymmetric ciphers.

- DPA (Differential Power Analysis): The key is extracted by exploiting variations in the power consumption caused by the key-dependent part of the **data flow**.
- Timing attacks: The key is extracted by recognizing key-dependent performance changes. As said before, most block ciphers don't have a key-dependent control flow, so this attack is not considered.

Power measurements are usually done by placing a shunt resistor in series to the power pins of the CPU, to measure the current absorbed in each time instant.
The key idea behind DPA is to guess (by brute force) small parts of the key and match each part independently from the others to the consumption measured from the device. This decreases the complexity to guess a key from $2^{L_{key}}$ to $N_{parts} \cdot 2^{L_{part}}$. Take a 128-bit key as an example: the brute force approach must try about $3.4 \cdot 10^{38}$ possible key values, while bytewise SCA tackles a $16 \cdot 2^8$ computational effort.
By having the power consumption measures available, the total number of possible key part values that should be tried is about $1,7 \cdot 10^{10}$. That is a 28 orders of magnitude improvement, meaning that those calculations can be done in reasonable time on a modern mobile phone.
Matching a (partial) key hypothesis with a trace from the oscilloscope involves the creation of a model of the power consumption while encrypting with that key and then matching that model to the traces. A simple but effective power consumption model is the Hamming weight of a specific partial value depending on both the key and the known value (plaintext or ciphertext).
As an example, consider an attack to the AES cipher, given the ciphertext. The most vulnerable point would be the last AddRoundKey operation, in which we know the resulting output and we can guess the subkey (and consequently the last state of the cipher). This can be done with an 8-bit granularity, if the key addition is done at each cell of the matrix representing the state. If the key addition is done in a single instruction for an entire column, we would need to consider 32 bits. The model can be fitted to real

values using a simple difference of means test or using Pearson's linear correlation coefficient.
This requires perfect knowledge of the timing of the interesting operation in the trace, but can be worked around by repeating calculations for each time instant and considering the highest correlation coefficient in any time point as the right one.
In order to reduce the noise, usually the output from the oscilloscope is filtered with a band-pass filter to keep (almost) only the interesting signal, which is synchronized to the device clock.

Measurement of the Consumption

The first step to mount the attack is to gather measurements of the actual power consumption of the device while computing a cipher. The power consumption can be either obtained by a power estimation tool such as the ones in common EDA tools, or recorded by means of a digital sampling oscilloscope.

The power consumption relative to an execution i is stored as a *power trace* $\mathbf{t_i}$, which can be viewed as a vector of M *power samples*, as shown in Equation 2.1.

$$\mathbf{t_i} = [t_{i,1}, t_{i,2}, \ldots, t_{i,M}] \tag{2.1}$$

Each power sample $t_{i,j}$ of a power trace $\mathbf{t_i}$ is the sum of different contributions, namely:

$$t_{i,j} = t_{i,j}^{\text{OP}} + t_{i,j}^{\text{DATA}} + t_{i,j}^{\text{NOISE}} + t_{i,j}^{\text{STAT}} \tag{2.2}$$

where $t_{i,j}^{\text{OP}}$ is the power consumption due to the specific operation executed, $t_{i,j}^{\text{DATA}}$ is the power consumption due to the data values being processed, $t_{i,j}^{\text{NOISE}}$ is the contribute coming from of environmental noise and $t_{i,j}^{\text{STAT}}$ is the static power consumption of the device. While $t_{i,j}^{\text{STAT}}$ is irrelevant to the purposes of power analysis (as it is constant and does not depend from what the platform is doing), what really matters in order to achieve good performance in the results is to minimize $t_{i,j}^{\text{NOISE}}$. $t_{i,j}^{\text{NOISE}}$ can be modeled as a random variable following a normal distribution $\mathcal{N}(0, \sigma)$, as it is not affected by the

ongoing operations. The other three contributions, for a fixed input and a fixed implementation platform, are constant. $t_{i,j}$ will thus follow a normal distribution $\mathcal{N}(\mu_{i,j}, \sigma)$, and it is thus possible to reduce the contribution of $t_{i,j}^{\text{Noise}}$ through averaging a reasonable amount of measurements of the same encryption. Clearly, if the power consumption of the device is predicted by an EDA tool the noise term is absent, as the tool is not able to predict the effective thermal and environmental noise which will affect the measurement setup.

Simple Power Analysis (SPA)

The most straightforward attack technique relying on the power consumption of a device is the Simple Power Analysis (SPA). SPA exploits the fact that, if at a specific point the control flow of a cryptosystem depends on the key, then the measurement of the dynamic power consumption of the circuit can leak the key. The most simple case is when an instruction is executed depending on a specific value of the secret key, as it happens in key dependent branches which can be found in the implementation of the straightforward square and multiply (or double and add) exponentiation (multiplication) algorithm. If the multiplication operation has a different power consumption from the squaring, it is possible for an attacker to distinguish them simply by looking at the recorded power trace of an exponentiation operation.

Differential Power Analysis (DPA)

Differential power analysis is a statistical power analysis technique, first introduced by P. Kocher et al. [15], that relying on the *difference of means* (DOM) statistical test to find the secret key stored in a device.

The fundamental difference between SPA and DPA attacks is that, SPA attacks exploit the difference in power consumption due to different key-dependent *operations* being executed while DPA attacks exploit the difference in power consumption due to the use of key-dependent *data*.

The main idea of DPA is to make predict the portion of the power consumption which depends on the key, for a small amount of key values, and employ the actual measurements to distinguish which one of these predictions is correct. To this end, a statistical test is employed to match the

measurements and the predictions: once the correct prediction is found, the value for a portion of the key is retrieved. Depending on the particular statistical test used to verify the hypotheses, DPA attacks take a different denomination in the literature.

Difference of Means - Common DPA Attack The first proposed method to validate the consumption prediction relies on a difference of means test. To employ this statistical tool, the attacker needs to classify the traces into two sets S_0, S_1, building them in such a way that the sample-wise mean consumption of the two sets differs significantly for some time instant j. To this end, the attacker follows the following procedure:

1. The attacker chooses a so-called *selection function*, that is a criterion to decide to which set a trace $\mathbf{t}_i$ belongs, relying on the value of the predicted power consumption $p_{i,l} = f_s(v_{i,l}) = f_s(f(d_i, k_l))$ with $1<i<N, 1 <l<|\mathcal{K}|$ previously computed with the data input d_i corresponding to the trace $\mathbf{t_i}$ and the key hypothesis k_l. A possible form of a selection function is the following

$$f_s(p_{i,j}) = \begin{cases} 1 \text{ if } p_{i,j}) > 4 \\ 0 \text{ if } p_{i,l} < 4 \end{cases} \tag{2.3}$$

2. The attacker employs the selection function to split the traces into two sets $S_{0,l}, S_{1,l}$ according to the value taken by it. This phase is repeated for every possible key hypothesis k_l, as they will generate different partitions of the traces.

3. For each of the possible partitions, the attacker computes the sample-wise mean $\mathbf{m}_{0,l}$ of all the traces belonging to $S_{0,l}$, and the sample-wise mean $\mathbf{m}_{1,l}$ of the ones belonging to $S_{1,l}$.

4. The attacker computes the sample wise differences $\delta_l = \mathbf{m}_{0,l} - \mathbf{m}_{1,l}$ for all possible key hypothesis k_l. If the key hypothesis is correct, the value of δ_l is expected to be significantly large for some time instant i, since the selection function has operated a correct partitioning of the traces into two sets where the mean consumption of the operation fits the predictions. If the key hypothesis is wrong, the selection function

simply operates a random partitioning of the traces into two sets, which are thus expected to have the same mean consumption.

Following this workflow, the attacker retrieves a part of the correct key and repeats the attack until the whole key is found, or the security margin (number of unknown bits) is low enough to be overcome by brute force. We note that there is no need to repeat the measurements (i.e. acquire a new set of traces $\mathbf{T}$) to do so: the same measurements can be employed successfully for the whole attack.

Pearson's Correlation Coefficient - Correlation Power Analysis (CPA) Correlation power analysis uses Pearson's (linear) correlation coefficient as a statistical test to distinguish the correct key hypothesis. Pearson's linear correlation coefficient describes how much two random variables can be expressed as one being in a linear relation with another. Pearson's linear correlation coefficient is obtained as dividing the value of the covariance between the two random variables under exam by the product of their variances, giving a result in the interval $[-1, 1]$. High values (regardless of the sign) of the correlation coefficient express a high correlation between the two variables, while a value close to zero indicates that they are not linearly correlated. Pearson's correlation coefficient between two random variables X and Y (commonly denoted as $\rho_{X,Y}$)is defined as

$$\rho_{X,Y} = \frac{\mathrm{cov}(X,Y)}{\sigma_X \sigma_Y} = \frac{E[(X-\mu_X)(Y-\mu_Y)]}{\sigma_X \sigma_Y}$$

In order to employ Pearson's correlation coefficient as a test for the correctness of a power consumption prediction, we will consider the measured power consumption in a precise time instant for all the traces $\mathbf{t}_j = [t_{1,j}, t_{2,j}, \ldots, t_{i,j}, \ldots, t_{N,j}]$ and its prediction for a fixed key hypothesis hypothesis $\mathbf{p}_l = [p_{1,l}, p_{2,l}, \ldots, p_{i,l}, \ldots, p_{N,l}]$ to be modeled by two random variables, of which the attacker needs to know if they are correlated. As the attacker does not know the theoretical distribution of these, but he only has samples from them available, he will need to employ the sample Pearson correlation coefficient as an estimator of the correct value of $\rho_{\mathbf{t}_j,\mathbf{p}_l}$. We recall that the sample Pearson correlation coefficient between the samples contained in $\mathbf{p}_l$ and $\mathbf{t}_j$ (commonly noted as $r_{\mathbf{t}_j,\mathbf{p}_l}$) can be computed as

$$r_{\mathbf{t}_j,\mathbf{p}_l} = \frac{\sum_i (t_{i,j} - \bar{t}_j)(p_{i,l} - \bar{p}_l)}{\sqrt{\sum_i (t_{i,j} - \bar{t}_j)^2 \sum_i (p_{i,l} - \bar{p}_l)^2}} \tag{2.4}$$

where $\bar{t}_j$ and $\bar{p}_l$ are the sample means over $\mathbf{t}_j$ and $\mathbf{p}_l$ respectively.

To check which key hypothesis is the correct one, the attacker computes the value of $r_{\mathbf{t}_j,\mathbf{p}_l}$ for all the time instants j and the key hypotheses k_l and checks which key hypothesis yields the highest peak correlation coefficient over the whole encryption.

The Pearson's correlation coefficient is typically considered an improvement over the distance of means that leads to more reliable and comparable results due to the use of a normalized covariance.

2.3.2 Active Attacks

Active attacks are based on the fact that a pair healthy-faulty of ciphertexts obtained from the same pair plaintext/key, leaks some information about the key itself. Faults are induced into circuits by:

- Power supply glitches
- EM pulses close to the circuit
- Clock signal glitches
- Ionizing Radiations

The techniques used to generate those faults are not always 100% reliable, so the fault might not be exactly in the intended point of the encryption.
There are some low cost fault injection methods (less than $3000) [6]. For instance, it's possible to run the chip with a depleted power supply, the attacker is able to insert transient faults starting from single bit errors and becoming more invasive as the supply voltage gets lower. Since this technique does not require precise timing, the faults tend to occur uniformly throughout the computation, thus requiring the attacker to be able to discard results that are not fit to lead an attack.
One refinement of the aforementioned technique is the injection of well-timed

power spikes or temporary brown-outs into the supply line of the circuit. Using this technique, it is possible to skip the execution of a single instruction in a software implementation of the cipher by reducing the feeding voltage for the duration of a single clock cycle. In order to inject a timed voltage lapse the attacker needs a custom circuit capable of dropping the feeding voltage synchronized with the circuit clock.
The temporal precision of the fault injection is directly dependent on the accuracy of the voltage drop both in terms of duration and synchronization with the target device.
Another viable option for an attacker is to tamper with the clock signal. For example, it is possible to shorten the length of a single cycle through forcing a premature toggling of the clock signal. Such shortening causes multiple errors corrupting a stored byte or multiple bytes. These errors are transient and thus it is possible to induce such faults without leaving any tamper evidence. To alter the length of the clock cycle, the attacker needs to have direct control over the clock line, which is the typical case when smart cards are targeted.
Another possibility for an attacker is to alter the environmental conditions, for instance, by causing the temperature to rise. A temperature rise has been reported to cause multiple multi-bit errors in DRAM memories.
There is a report of a thermal fault injection attack against the DRAM chips of a common desktop computer through the use of a 50W light bulb and a thermometer. Drawbacks of this technique are the invasive faults and the potential permanent damage of the device.
A practical way to induce faults is to cause strong EM disturbances near it. The currents induced in the circuit by strong EM pulses cause temporary alterations of the level of a signal, which may be recorded by a latch. The spark generator can be a simple piezoelectric gas lighter. All the parts of the circuit which do not need to be disturbed should be shielded.
Assuming the attacker is able to successfully decapsulate a chip, he can perform fault injection attacks by illuminating the die with a high energy light source such as an UV lamp or a camera flash. The strong radiation directed at the silicon surface can cause the blanking of erasable EPROM and FLASH memory cells where constants needed for an algorithm execution are kept.

A class of threats which cannot be ignored if the attackers have access to a larger budget includes fault injection techniques that rely on having a direct access to the silicon die.
One of these techniques is based on the use of a strong and precisely focused light beam to induce alterations in the behavior of one or more logic gates of a circuit. A strong radiation of a transistor may form a temporary conductive channel in the dielectric, which, in turn, may cause the logic circuit to switch state in a precise and controlled manner (provided that the used etching technology is not too small).
In order to obtain a sufficiently focused light beam from a camera flash, a precision microscope must be used. The main limitation of this technique is the non-polarized nature of the white light emitted by the camera flash resulting in scattering of the light when focused through non-perfect lenses.
The most straightforward refinement of the previous technique is to employ a laser beam instead of a camera flash. The injected fault model is similar to that obtained when using a concentrated light beam.
Those techniques include focused light beams or laser beams which can create a temporary conductive channel in the dielectric, bypassing the hardwired logic. The most accurate and powerful fault injection technique uses Focused Ion Beam stations (FIB) that enable an attacker to arbitrarily modify the structure of a circuit, reconstruct missing buses, cut existing wires, mill through layers and rebuild them.
Active attacks are not handled in this work, but it is possible to expand this work to deal with them.

Active attacks are described with less details than passive attacks, as they are out of scope for this work.

2.4 Software Protection from Side Channel Attacks

The basic idea behind countermeasures is to break the dependency between the computation of a cryptographic algorithm and the information leaked by the side-channels. There are basically two kind of countermeasures that are called *hiding* and *masking*. The former tries to hide the information leaked without altering the computation, while the latter tries to mask the computation without altering the side-channels.

Countermeasures can be implemented at different levels of abstraction. At very low level, countermeasures are implemented by using protected logic styles that try to hide the normal power consumption. At the architectural level, the order of instruction can be randomized or dummy instruction can be inserted randomly in order to obtain power traces that are not directly comparable. Finally, at algorithm level, the cryptographic algorithm can be altered in such a way that the information leaked is not correlated with the expected intermediate results of a normal computation [17].
In order to achieve an higher the level of security the countermeasures can be also mixed and integrated for instance by implementing hiding countermeasures after that masking countermeasures have taken place. A common drawback in countermeasures is that they typically come at the price of a speed loss, higher chip area or higher power consumption that make countermeasures hard to implement in the practice.
The first work tackling the problem of automatically protect software implementations of cryptographic algorithm from power analysis attacks is presented by Bayrak *et al.* in [7]. The authors identify the instructions which are most vulnerable to power analysis running their target implementation and profiling the power consumption of the underlying platform.
In this way, they identify the most vulnerable clock cycles of the program execution and associate to each of them the corresponding assembly instruction together with a *sensitivity value*.
Instructions, whose sensitivity is greater than a chosen threshold are replaced by an appropriate code snippet, which realizes a random pre-charging of either the registers or the memory cells.
Note that even if the proposed workflow is general enough, the implemented code transformation step is specific for devices whose power consumption is proportional to the Hamming Distance between two consecutive execution cycles.
Moreover, the proposed approach needs a prototype device to be profiled for leakage.

2.4.1 Hiding

The basic idea of hiding is to remove the dependency of the computation from the power consumption by altering the power consumption of the de-

vice. This is can be realized for instance by making the power consumption either constant or random.
The hiding techniques affect either the time or the amplitude dimension [17]. In the former case, the operations of the algorithm are executed at different moments of time for each different execution, while in the latter case the power consumption of each operation is altered randomly.
To the class of countermeasures that affect the time dimension it does belongs the countermeasures which randomly insert or shuffle the order of operations every execution. To the class of countermeasures that affect the amplitude dimension it does belongs the countermeasures which introduce noise in the form of switching activity in the normal cryptographic computation in order to reduce the information leaked about the computation.
The shuffling technique involves computing the required values at different, randomized, times in each encryption, in order to reduce the correlation coefficient.
Inserting dummy instructions into the compiled code helps to create confusion in power profiles between the real instructions and the power consumption caused by the dummy, randomized, ones.

2.4.2 Masking

A very common countermeasure to protect cipher implementations against SCA is to randomize the way sensitive variables are computed through *masking* techniques [13, 17]. The principle is to add one or more random values (*masks*) to every sensitive intermediate variable occurring during the computation. In a masked implementation, each sensitive intermediate value is represented as split in a number of *shares* (containing both the randomized sensitive value and the masks employed), which are then separately processed. To this end, the target algorithm is modified to process each share and recombine them at the end of the computation. This technique effectively hinders the attacker from formulating a correct power consumption model, as the instantaneous power consumption is independent from the processed value. Typically, masking techniques are categorized by the number of masks d employed for each sensitive value, which is known as *order* of the masking. A d-th-order masking can always be theoretically broken by

a $(d+1)$-th-order attack, i.e. an attack exploiting the combination of $d+1$ measurements of different instructions, during an execution, to build a mask-independent prediction [17, 22, 23]. In practice, the difficulty of carrying out a d-th-order attack increases exponentially with d, due to the difficulty of guessing which time instant is the one when the sensitive computations happen [8]. Even though a high order masking is crucial to ensure good security margins, only a few $d > 1$ order masking schemes exist. Moreover, masking schemes are developed for specific ciphers, leading to the hand-crafting of a whole suite of protected ciphers. The current state-of-the-art methodology to develop power analysis-resistant encryption primitives is to manually implement masking schemes in assembly code, applying them to all the instructions. This is usually performed tackling the issue of adding masking scheme to each cipher with ad-hoc, per cipher, techniques. [13, 17, 22, 23].

2.4.3 Automated Protection

Currently, one of the efforts towards an automation is provided by Moss *et al.* [18] proposed a first attempt at automating the process of inserting a 1st order masking scheme in the code of AES using an *ad-hoc* translator. Their scheme relies primarily on *type inference*, a kind of static analysis which is strongly dependent on the source language. To this end, the authors of [18] designed their own Domain Specific Language (DSL) with a specialized type system, which allows type inference. In practice the DSL source code must contain an explicit annotation for variables to be protected (depending on the programmer choices), as there is no automatic evaluation of the security margin bound to each instruction of the program to be executed. In addition, from a practical point of view, it is worth noting that most encryption primitives, especially for application in embedded systems, are not developed with a DSL, being instead available primarily in C.

Chapter 3

Side-Channel Vulnerability Analysis and Countermeasures Application

This chapter contains a description of the Security-oriented Data Flow Analysis (SDFA), a specifically designed data flow analysis but has the objective of analyzing data dependencies with respect to an encryption key. Then it explains how this algorithm can be used to determine the exact attack surface of a block cipher, through automatically recognizing the user key, the key material produced by the key scheduler and the order in which this key material is used. Subsequently the SDFA is used to gain knowledge about propagation of this key material, and this information is further condensed into an *instruction resistance* value, considering the best options available to an attacker. This *instruction resistance* value is then used to apply selective masking, by adding a pass to the LLVM framework.

3.1 Security Oriented Flow Analysis

The source-language code is extended with custom attributes to allow the developer to provide information to the compiler about the variables containing the *key material* and the *plaintext data* (f.i., the GNU extension mechanism for the C language). The decorated source program is parsed by the *Front-End* to produce the IR, which is optimized using standard

optimization passes (f.i., the `-O2` option of either `clang` or `gcc` compiler). The optimized IR is analyzed by a new *Security-oriented Data-Flow Analysis* (SDFA) pass, which adds metadata to each defined variable to identify its level of vulnerability. The SDFA can also identify control flow issues that would prevent a precise analysis and protection of the code, which actually needs to be fixed by the programmer. The vulnerability information is then used by the *Masking Application* pass, which modifies the normalized IR code through applying masking countermeasures where appropriate. The output IR is then translated to the target assembly by the standard *Back-End* pass.

3.1.1 Forward

To define a *security-oriented data-flow analysis*, aimed at detecting the amount of key material involved in the computation of an intermediate value, a CFG built from an IR in SSA form will be considered. The goal of this analysis is to identify a set of nodes of the CFG representing the portions of the program amenable to passive side-channel attacks. The choice of building a single-instruction-per-node CFG is justified by the fact that the application of countermeasures implies a significant performance penalty and should be done sparingly. An instruction is deemed to be vulnerable if computing a model of its behavior for each possible value of the key bits from which its output value depends is computationally feasible. Computing the aforementioned model is the ground on which passive side-channel attacks are built, as its predictions are matched against the measured behavior of the considered device. DFA aims at gathering information about the possible set of values calculated at each statement of a program, employing the CFG to determine the propagation paths of each computed value [14, 19]. In our case, the information to be traced is the data-dependence between any bit computed by a program instruction and every bit of the cryptographic key. Such a choice is mandated by the need to consider possible side-channel attack models predicting the behavior of the computation of a single bit within a w-wide value [15]. The aforementioned relation is modeled through an n-bit Boolean lattice $(\mathbb{BV}^n, \sqcup, \sqcap)$, where the elements of the support set

$$\mathbb{BV}^n = \{v_0, \ldots, v_{2^n-1}\} = \{\langle 00\ldots0\rangle, \ldots, \langle 11\ldots1\rangle\}$$

represent all the possible combinations of key bits from which a bit of an intermediate result depends on, thus n equals the key size of the cipher under exam. The bottom of the lattice $\bot$ is represented by the element $\langle 0 \dots 0 \rangle$, which indicates that no key bits are involved, while the top $\top$ element is $\langle 1 \dots 1 \rangle$, denoting that all the key bits are involved. The characteristic partial order relations $\succeq$ and $\preceq$ over the lattice elements are defined as follows:

$$\begin{aligned} v \succeq v' &\Leftrightarrow \exists\ v'' \,|\, v' \sqcup v'' = v \\ v \preceq v' &\Leftrightarrow \exists\ v'' \,|\, v' \sqcap v'' = v \end{aligned} \qquad v, v', v'' \in \mathbb{BV}^n$$

with the $\sqcup$ operation being defined as the common bitwise inclusive-*or*, and the $\sqcap$ operator being the bitwise *and*.

Our SDFA computes how many key bits are involved in the computation of each intermediate value, i.e., due to the SSA nature of the IR from which the graph is obtained, how many bits are involved in the output of each bit composing the outcome of any instruction `I` of the program. To this end, the key propagation is computed for every bit of any size(`I`)-bit wide intermediate result through associating a *leakage vector* $V_{\mathtt{I}}=(v_{\text{size}(\mathtt{I})-1}, \dots, v_t, \dots, v_0)$ of size(`I`) elements $v_t \in \mathbb{BV}^n$ to each node of the CFG, which represents a single SSA instruction. Each v_t represents the key bits involved in the computation of the t-th bit of the corresponding intermediate value output by `I`. with t ranging from 0 to size(`I`), i.e. from the least to the most significant bit of the instruction outcome.

The *meet* and *join* operations on the leakage vectors (denoted as $\vee$ and $\wedge$, respectively) are defined as the extensions of the aforementioned $\sqcup$ and $\sqcap$. Given two leakage vectors $V_{\mathtt{I}}=(v_{s-1}, \dots, v_0)$ and $V_{\mathtt{J}}=(v'_{s-1}, \dots, v'_0)$ of equal size s=size(`I`)=size(`J`), the *meet* composition law between $V_{\mathtt{I}}, V_{\mathtt{J}} \in (\mathbb{BV}^n)^{\mathrm{s}}$ is defined as $V_{\mathtt{I}} \vee V_{\mathtt{J}}=(v_{s-1} \sqcup v'_{s-1}, \dots, v_0 \sqcup v'_0)$. Dually,the *join* composition law is defined as: $V_{\mathtt{I}} \wedge V_{\mathtt{J}}=(v_{s-1} \sqcap v'_{s-1}, \dots, v_0 \sqcap v'_0)$.

Using the information provided by the leakage vector, it's possible to introduce a definition of instruction resistance. Given an instruction, its resistance to passive SCA is formally defined as follows:

Definition 3.1.1 (Instruction Resistance). Consider an IR instruction `I` with

a size($\mathtt{I}$)-bit output value, and the associated leakage vector

$$V_{\mathtt{I}} = (v_{\text{size}(\mathtt{I})-1}, \ldots, v_t, \ldots, v_0) \in (\mathbb{BV}^n)^{\text{size}(\mathtt{I})}$$

Denoting the Hamming weight of a bit-vector $v_t \in V_{\mathtt{I}}$ as $\text{HW}(v_t)$, the instruction resistance is defined as:

$$\min_{v_t \in V_{\mathtt{I}} : v_t \neq \perp} \{\text{HW}(v_t), +\infty\}$$

that is, the minimum number of key bits influencing a bit of the output value of a sensitive $\mathtt{I}$. An instruction that does not depend on any key bit is conventionally associated to a resistance value equal to ∞.

To automatically evaluate the resistance of an instruction `I`, it is necessary to consider the leakage vector associated to each instruction preceding it and take into account which definitions are used by it. This information is captured by the notion of *In-Set* of the instruction. The propagation of resistance information through the specific transformation operated by `I` is captured by the notion of *Out-Set* of the instruction.

Definition 3.1.2 (In-Set). Given an instruction $\mathtt{I}$, the input set $in(\mathtt{I})$ is defined as the set of the leakage vectors associated to all the immediate predecessors of $\mathtt{I}$ on the CFG $\mathcal{G}(\mathcal{B}, \mathcal{E})$:

$$in(\mathtt{I}) \stackrel{def}{=} \left\{ V_{\mathtt{J}} \mid \mathtt{J} \in \mathcal{B}, \mathtt{J} \in pred(\mathtt{I}), V_{\mathtt{J}} \in (\mathbb{BV}^n)^{\text{size}(\mathtt{J})} \right\}$$

Definition 3.1.3 (Out-Set). Given an instruction $\mathtt{I}$, the output set $out(\mathtt{I})$ is defined as the set of the leakage vectors associated to every immediate predecessor of $\mathtt{I}$ on the CFG $\mathcal{G}(\mathcal{B}, \mathcal{E})$ plus the one of $\mathtt{I}$, $V_{\mathtt{I}} \in (\mathbb{BV}^n)^{\text{size}(\mathtt{I})}$:

$$out(\mathtt{I}) \stackrel{def}{=} \{V_{\mathtt{I}}\} \cup \left\{ V_{\mathtt{J}} \mid \mathtt{J} \in \mathcal{B}, \mathtt{J} \in pred(\mathtt{I}), V_{\mathtt{J}} \in (\mathbb{BV}^n)^{\text{size}(\mathtt{J})} \right\}$$

3.1.2 Local Security-Oriented DFA

Definition 3.1.4 (Local Security-oriented DFA).
Each instruction $\mathtt{I}$ within a basic block is characterized by an opcode, $op(\mathtt{I})$, an In-set: $in(\mathtt{I})$, and an Out-Set: $out(\mathtt{I})$. The effect of the execution of $\mathtt{I}$ is modeled through a transformation function $\mathcal{F}_{op(\mathtt{I})}(\cdot)$ taking as input its

In-set. Therefore, for any instruction I the following equations can be stated:

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

The SDFA solves the set of simultaneous equations derived from the instructions in the basic block through subsequent approximations until a fixed-point is reached. In the particular case of the local SDFA, the convergence is achieved in a single step. The behavior of the transformation function depends on the opcode of the instruction, as the propagation of the key dependencies depends on its nature. Note that, the semantics of each instruction determine also the bit-size (size(I)) of its output value. Thus, to compute the corresponding Out-Set it may be necessary to produce a properly sized leakage vector. We denote as $\textsc{resize}_{\mathtt{I}}(V_{\mathtt{J}})$ the adaptation of the leakage vector $V_{\mathtt{J}}$ to the same size of the instruction I as follows:

$$\textsc{resize}_{\mathtt{I}}(V_{\mathtt{J}}) = \begin{cases} (v_{\text{size(I)}-1}, \ldots, v_t, \ldots, v_0), \\ v_t \in V_{\mathtt{J}}, \quad \text{if size(I)} \leq \text{size(J)}; \\ (\perp_{\text{size(I)}-\text{size(J)}-1}, \ldots, \perp_{\text{size(J)}}, \ldots, v_t, \ldots), \\ v_t \in V_{\mathtt{J}}, \quad \text{if size(I)} > \text{size(J)} \end{cases}$$

Let $\textsc{operands}(\mathtt{I})$ be the set variables used by I as operands. As the IR is in SSA form, each variable is defined only once, so, with a small notation abuse, the form $\mathtt{J} \in \textsc{operands}(\mathtt{I})$ will be used to denote that instruction J defines one of the operands of I. For each instruction class of the IR, our analysis assumes the transformation function to be defined as:

$$out(\mathtt{I}) = \mathcal{F}_{op(\mathtt{I})}(in(\mathtt{I})) \stackrel{\text{def}}{=} in(\mathtt{I}) \cup \{V_{\mathtt{I}}\}$$

where the leakage vector of the current instruction $V_{\mathtt{I}}$ is computed according the formulae presented hereafter.

Arithmetic, Bitwise-logic and `cmp` instructions

These instructions can be partitioned in two sets depending on the computation of their leakage vectors.

The first set includes all instructions with an opcode, $op(\mathtt{I})$, specifying

a bitwise operation (f.i., `not`, `and`, `or`, `xor`), an `add` or `sub` operation, with the exception of the `and` and `or` with an immediate operand, as well as `shift`, `zero-` and `sign-extension`. The evaluation of the leakage vector bound either to an `add` or to a `sub` operation is done through considering them as a `xor` operation. This assumption neglects the influence of the carry/borrow propagation in the computation of the result. This is justified by the fact that the most favorable situation for an attacker is when there is no carry propagation (i.e., when the influence of the key bits on each bit of the final outcome is minimized). The computation of the leakage vector of any of the aforementioned instructions is the composition of the leakage vectors in their In-Sets, so that the output bit dependencies (from the key bits) are the ones of the corresponding bits of the input operands added together: $V_{\mathtt{I}}{=}\bigvee_{\mathtt{J}\in\text{OPERANDS}(\mathtt{I})} \text{RESIZE}_{\mathtt{I}}(V_{\mathtt{J}})$. The second set of instructions includes `mul`, `div`, `mod`, and `cmp` operations. Multiplication, division and modulo operations diffuse the information contained in the operand bits, so that every bit of the output depends on every bit of the inputs. Let `I` be any of these instructions, and let $\mathtt{J}\in$OPERANDS(`I`) be the instructions computing the operand values of `I`, with $V_{\mathtt{J}}{=}(v_{\text{size}(\mathtt{J})-1}, \ldots, v_t, \ldots, v_0)$ being the corresponding leakage vectors. The leakage vector of the instruction result is computed so that, for each of its bits, the dependencies (from the key bits) of all operands bits are added together: $V_{\mathtt{I}}{=}\bigvee_{\mathtt{J}\in\text{OPERANDS}(\mathtt{I})} \text{RESIZE}_{\mathtt{I}}(\widehat{V}_{\mathtt{J}})$, where $\widehat{V}_{\mathtt{J}} = (\widehat{v}_{\text{size}(\mathtt{J})-1}, \ldots, \widehat{v}_t, \ldots, \widehat{v}_0)$, $\forall\ t \mid \widehat{v}_t = \bigsqcup_{0\le t<\text{size}(\mathtt{J})} v_t$. Note that, when considering a `cmp` instruction the outcome computed by the instruction is reduced to a single bit.

Bitwise `and` and `or` instructions with an immediate operand

Denoting as imm_i the i-th bit ($0{\le}\ i{<}\text{size}(imm)$) of the immediate operand, define a support leakage vector V_{imm} as:

$$V_{imm} = \begin{cases} (\ldots, \langle\ imm_i, \ldots, imm_i\rangle, \ldots), \\ \qquad \text{with } 0 \le i < \text{size}(imm),\ \text{if } op(\mathtt{I}) = \mathtt{and} \\ (\ldots, \langle \neg imm_i, \ldots, \neg imm_i\rangle, \ldots), \\ \qquad \text{with } 0 \le i < \text{size}(imm),\ \text{if } op(\mathtt{I}) = \mathtt{or} \end{cases}$$

to model the dependency-cancelling effect of the absorbing elements of bitwise `or` and `and` operations (1 and 0, respectively) on the input leakage vector

$V_{\mathtt{J}}$. The output leakage vector $V_{\mathtt{I}}$ is thus obtained removing the cancelled dependencies from the input ones as follows:

$$V_{\mathtt{I}} = \text{RESIZE}_{\mathtt{I}}(V_{imm}) \wedge \bigvee_{\mathtt{J} \in \text{OPERANDS}(\mathtt{I})} \text{RESIZE}_{\mathtt{I}}(V_{\mathtt{J}})$$

`shift` instructions with an immediate operand

Let $\mathtt{J} \in$ OPERANDS(I) be the instruction producing the non-immediate operand of I, and $V_{\mathtt{J}}=(v_{\text{size}(\mathtt{J})-1}, \ldots, v_0)$ the corresponding leakage vector. The leakage vector associated to I is $V_{\mathtt{I}}=\text{RESIZE}_{\mathtt{I}}(\widehat{V})$:

$$\widehat{V} = \begin{cases} (v_{\text{size}(\mathtt{J})-1-imm}, \ldots, v_0, \perp_{imm}, \ldots, \perp_0), \\ \qquad \text{if } op(\mathtt{I}) = \mathtt{shl}, \mathtt{ashl} \\ (\perp_{\text{size}(\mathtt{J})-1}, \ldots, \perp_{\text{size}(\mathtt{J})-1-imm}, v_{\text{size}(\mathtt{J})-1}, \ldots, v_{imm}), \\ \qquad \text{if } op(\mathtt{I}) = \mathtt{shr}, \mathtt{ashr} \end{cases}$$

The computation of $\widehat{V}$ takes into account the fact that the bits of the output are a permutation of the input ones, possibly discarding some.

Data-dependent `shift` instructions

In this case the non-immediate operands simply considering the outcome of the instruction as an unpredictable result. The corresponding leakage vector is conservatively estimated through removing every dependence from the key bits: $V_{\mathtt{I}}=(\perp_{\text{size}(\mathtt{I})-1}, \ldots, \perp_0)$.

`store` instruction

`store` operations do not produce any new value. Thus, the following equation applies: $out(\mathtt{I}) = in(\mathtt{I})$ as there is no leakage vector.

`load` instruction

The operands of the `load` instruction can compute an address value that possibly depends on the key bits. Thus, every output bit is considered as dependent on every bit of the address.

Given $\mathtt{J} \in$ OPERANDS(`load`), with $V_{\mathtt{J}} = (v_{\text{size}(\mathtt{J})-1}, \ldots, v_0)$, the information leakage is:

$$V_{\mathtt{load}} = \bigvee_{\mathtt{J} \in \text{OPERANDS}(\mathtt{load})} \text{RESIZE}_{\mathtt{I}}(\widehat{V}_{\mathtt{J}})$$

where

$$\widehat{V}_{\mathtt{J}} = (\widehat{v}_{\mathrm{size}(\mathtt{J})-1}, \ldots, \widehat{v}_t, \ldots, \widehat{v}_0),\ \forall\ t \mid \widehat{v}_t = \bigsqcup_{0 \leq t < \mathrm{size}(\mathtt{J})} v_t$$

If the address depends on the key and the loaded value also contain some key material the above leakage vector is used as a conservative approximation of the actual one.

`zero-` and `sign-extension` instructions

These two instructions are usually employed in an IR when a change of data type occurs. In this respect, each of them can be managed as an instruction, `I`, with a non-immediate operand that must be extended up to known size. Let $\mathtt{J} \in \mathrm{OPERANDS}(\mathtt{I})$ be the instruction producing the non-immediate operand of `I`, and $V_{\mathtt{J}}=(v_{\mathrm{size}(\mathtt{J})-1}, \ldots, v_0)$ the corresponding leakage vector. The data associated to the instruction are simply computed as:

$$V_{\mathtt{I}} = \begin{cases} (\bot_{\mathrm{size}(\mathtt{I})-1}, \ldots, \bot_{\mathrm{size}(\mathtt{J})}, v_{\mathrm{size}(\mathtt{J})-1}, \ldots, v_0), \\ \qquad \text{if } op(\mathtt{I}) = \texttt{zero-extension} \\ (v_{\mathrm{size}(\mathtt{I})-1}, \ldots, v_t, \ldots, v_{\mathrm{size}(\mathtt{J})}, v_{\mathrm{size}(\mathtt{J})-1}, \ldots, v_0), \\ \text{with } v_t = v_{\mathrm{size}(\mathtt{J})-1},\ \text{where } \mathrm{size}(\mathtt{J}) \leq t < \mathrm{size}(\mathtt{I}), \\ \qquad \text{if } op(\mathtt{I}) = \texttt{sign-extension} \end{cases}$$

3.1.3 Global Security-Oriented DFA and Control Flow Normalization

Given the local SDFA, it is possible to construct a global SDFA through extending the data-flow equations to cover the case where an instruction `I` has multiple immediate predecessors (i.e., $|pred(\mathtt{I})|>1$). These instructions can be easily identified since the SSA IR marks the confluence points by a ϕ-function, i.e. a marker pointing out which are the actual output values of the predecessors of the instruction which should be used. To this end, the relation between the In-set $in(\mathtt{I})$ of each instruction with multiple predecessors and its Out-set, combining the contribution of the $pred(\mathtt{I})$ is defined through the so-called *confluence* operator. In data-flow analysis techniques, the *confluence* operator is employed to obtain a conservative information regarding the data-flow, as it is not possible to fully predict which value among the ones present in the out-sets of $pred(\mathtt{I})$ will be employed by `I`. This is obtained through preserving only the data-flow information common to all the

incoming execution paths, that is, applying a so-called *meet-over-all-paths* policy. In our context, it's possible to derive the information associated to the output of an instruction with multiple predecessors through combining them with the *meet* operation on the leakage vectors. More formally our global SDFA is defined as follows.

Definition 3.1.5 (Global Security-oriented DFA).
Let $\mathcal{G}(\mathcal{B},\mathcal{E})$ be a control-flow graph and let $\mathit{I}\in\mathcal{B}$ be an instruction with $|pred(\mathit{I})|\geq 1$, then the equations defining its In-set and Out-set are given as follows:

$$\begin{aligned} in(\mathit{I}) &= \begin{cases} \emptyset, & \text{if } pred(\mathit{I})=\emptyset \\ \bigcup\limits_{\mathit{H}\in\mathcal{B}} \left\{ \bigwedge\limits_{V_{\mathit{H}}\in out(\mathit{J}),\, \mathit{J}\in pred(\mathit{I})} V_{\mathit{H}} \right\}, & \text{otherwise} \end{cases} \\ out(\mathit{I}) &= \mathcal{F}_{op(\mathit{I})}(in(\mathit{I})) \end{aligned}$$

The global SDFA defined above, while theoretically correct, is not of practical use "as is": the goal is to have an accurate assessment of the vulnerability, as well as to provide low overhead countermeasures. the issue lies in the fact that, whenever a confluence is computed where information would be lost due to a loop edge (i.e., for some ϕ-function instruction that follows a confluence between with a back-edge in the CFG), we have that:

$$\exists\ \mathtt{J} \in \textsc{operands}(\mathtt{I}) \text{ s.t. } V_{\mathtt{J}} \neq \bigwedge_{\mathtt{H}\in\textsc{operands}(\mathtt{I})} \textsc{resize}_{\mathtt{I}}(V_{\mathtt{H}})$$

which may underestimate the effective amount key bits influencing the result, leading to a sub-optimal analysis in terms of precision. To prevent this, the SDFA is interrupted, and the Control Flow Normalization pass is invoked to perform a *loop peeling* [5] action, extracting one iteration from the loop, thus delaying the confluence to the next iteration, and the SDFA is restarted. At the end of this process, there are two possible situations: a fully unrolled loop, or a condition where all the definitions used in the ϕ instruction carry the same information – and, if the encryption algorithm is correctly designed, the dependency from all the key bit pertains to a large amount of live variables. By contrast, if there are some key bits, which are never involved in data dependencies with the live variables, the encryption algorithm does not

fully employ the key to protect the plaintext. After performing the global SDFA, it's possible to identify with precision the amount of key bits influencing the computation of each intermediate value of the algorithm, thus in turn it's possible to decide which ones should be the target of the countermeasure application. Another concern for the precision of the proposed SDFA is represented by conditional statements, for instance, consider the following conditional expression in C language: `res=(r<0) ? r^c : r;`. Denoting as `%r` and `%c` the virtual registers for the variable `r` and `c`, the previous C statement is translated in SSA IR form as shown in the left-pane of the following code snippets:

```
%1=icmp slt i8 %r,0
%2=xor i8 %r,i8 %c
%res=select i1 %1,i8
%2,i8 %r
```

```
%1=ashr i8 %r,7
%2=xor i8 %r,i8 %c
%res=xor i8
%res,%r
```

The IR on the left, is transformed by an *if-conversion* pass in the one on the right, which contains an equivalent sequence of instructions. Note that the comparison with the zero value (checking if `%r` is negative) is substituted with the arithmetic right shift to recover the sign bit of the variable `r`.

As examined block ciphers have a really simple control flow, the global SDFA and control flow normalization pass were not automated in this implementation and their effects were obtained by manually unrolling loops with the preprocessor.

3.1.4 Backward Security-Oriented DFA

As it is typical to attack cryptographic algorithms through predicting an intermediate value of the computation preceding the output (the known ciphertext), here will be described the Backward SDFA algorithm to identify the resistance of instructions in this case.

Backward DFAs are constructed similarly to the forward ones, through reversing the relation between In-set and Out-set. However, our SDFA is tailored to the approach used by the attacker to find out portions of the cryptographic key. In the general case, the backward DFA combines the information from the In-sets of the successor of an instruction to compute its

Out-set, while the In-set of the instruction is obtained by applying an $\mathcal{F}_{op(\mathtt{I})}$ function to the Out-set. However, when the key is directly combined with the state of the algorithm, it adds its protection to the other inputs of the combining instruction. This is taken into account by having $\mathcal{F}_{op(\mathtt{I})}$ functions that combine not only the Out-set, but also the information coming from the instructions that define the values used by the instruction. The general form of the Backward SDFA equations is as follows:

$$\begin{aligned} out(\mathtt{I}) &= \begin{cases} \emptyset, & \text{if } succ(\mathtt{I})=\emptyset \\ \bigcup\limits_{\mathtt{H}\in\mathcal{B}} \left\{ \bigwedge\limits_{V_\mathtt{H}\in in(\mathtt{J}),\, \mathtt{J}\in succ(\mathtt{I})} V_\mathtt{H} \right\}, & \text{otherwise} \end{cases} \\ in(\mathtt{I}) &= \mathcal{R}_{op(\mathtt{I})}(out(\mathtt{I})) \end{aligned}$$

For the arithmetic-logic instructions, the definition of the transformation function $\mathcal{R}_{op(\mathtt{I})}$ is given by: $in(\mathtt{I})=\mathcal{R}_{op(\mathtt{I})}(out(\mathtt{I}))\stackrel{\text{def}}{=}out(\mathtt{I}) \cup \{V_\mathtt{I}\}$, where $V_\mathtt{I}=\bigvee_{\mathtt{J}\in \mathrm{USE}(\mathtt{I})} \mathrm{RESIZE}_\mathtt{I}(V_\mathtt{J})$. The main difference from the forward SDFA is the case where one of the operands of `I` contains some key material. Accordingly, the transformation function is modified as follows:

$$in(\mathtt{I}) = \mathcal{R}_{op(\mathtt{I})}(out(\mathtt{I})) \stackrel{\text{def}}{=} out(\mathtt{I}) \cup \{V_\mathtt{I} \vee \mathrm{RESIZE}_\mathtt{I}(V_\mathtt{K})\}$$

where $V_\mathtt{K}=(v_{\mathrm{SIZE}(\mathtt{K})-1}, \ldots, v_t, \ldots, v_0)$ is the contribution of each bit of the key (v_t denotes a bit vector that has a single bit set at position t from the least to the most significant). This modification takes into account the fact that an attacker will need to make an hypothesis on the whole key material involved either directly (i.e. as an operand) or indirectly (i.e. in the computation of values depending on the result) with the instruction under exam. Thus, the Backward SDFA has one forward path (the one regarding the use of the key), an uncommon, but not unheard of case for DFA [14]. As the Backward SDFA aims at finding the dependence relations considering the point of view of an attacker in possess of the outputs of the algorithm, the relations among the key bits imposed by the key schedule should be considered in reverse order. To this end, the analysis considers the last values produced by the key schedule as the actual *initial* key and derives the relations with the rest of the key material backwards. The key schedule is identified as the set of instructions that employ, directly or indirectly, only key material and no plaintext. After identifying such values, a visit of the whole cipher CFG,

starting from the instructions computing the output, is performed to identify instructions that use the key schedule values. The first key schedule values to be found are marked as the *initial* key. The amount of these values which should be considered as *initial*, lest the whole key schedule be marked, is bound by their total size being at least the size of the user key, and by them being dependent on all the user key bits. This last condition can be easily checked as a full key schedule forward DFA has been performed at the beginning of this *initial* key identification step. After these *initial* key values are identified, the Backward SDFA computes the dependence from them as imposed by the data-flow equations.

3.2 Automated Vulnerability Analysis

The algorithm described in this section aims at determining the effective *attack surface* of a block cipher in terms of vulnerable operations and the corresponding effort needed to lead a side channel attack.

To this end, it is important to understand which parts of the code are sensitive to the attack. The sensitive instructions are those that mix plaintext material and key material, i.e. those intermediate values that can be predicted based on knowing the plaintext and guessing the involved part of the key material. The complexity of the side channel attack rises exponentially in the number of bits to be guessed when computing the hypothetical values of the side channel.

In order to ease the reader's understanding of this argument, let's describe a toy cipher that will be used to describe examples alongside the description in the next section. The toy cipher is a four round block cipher with a 32 bit state and a 64 bit key. The round function is a simple left bitwise rotation of the 32 bit state and the addition via exclusive or of a word of the key material produced by the key schedule.

The key schedule takes as input the 64-bit encryption key and yields four 32-bit words of key material. The first two words are obtained as the first and second half of the encryption key, whereas the third one is obtained as the exclusive or of the first two words, and the last word is obtained by applying a shift and a xor mask to the second word.

The code reported in Figure 3.1 includes the attributes `plain` and `key`

attached respectively to the plaintext and user key and employed by our framework to keep track of those values in the IR.

Clang translates the aforementioned C code to IR code as reported in Figure 3.2. Figure 3.2 shows the IR code corresponding to each source code statement from line 12 to line 27 — the remaining lines, which compose the epilogue and prologue of the cipher function are omitted for the sake of clarity. The IR code defines the Data Dependency Graph (DDG) reported in Figure 3.3. Since the source code is fully unrolled and has no control flow, the DDG is semantically equivalent to the IR code, save for the opcodes which are omitted in the figure for the sake of readability.

It is worth noting that the IR code follows the SSA form, thus each virtual register is defined only once. The names of the virtual registers have been modified for the sake of readability, by using meaningful names (the compiler would just employ a progressive numbering). In particular, virtual registers defined in the round function are named by the operator that produces them and the round index, while the virtual registers corresponding to the key schedule are named as `%sk`i, where i is the index of the key schedule word.

The key material itself is obtained from the original encryption key through applying a key schedule procedure which expands the encryption key into a fixed number of *round keys*, each used in a different iteration of the round function of the cipher.

It is important to distinguish instructions that operate directly on a round key (or part of it) from those that operate on intermediate values computed by previous instructions as a combination of key material and plaintext. The first step to mount a side channel attack is to select which intermediate values resulting from the computation of the plaintext and key material values should be predicted.

Definition 3.2.1 (Directly employed key material). Directly employed key material is defined as the value of a round key or part of it used as input to an instruction that has another input, which is either an intermediate value or (part of) the plaintext.

In the toy cipher example, the `%sk0`, `%sk1`, `%sk2` `%sk3` nodes output the directly employed key material. It is also necessary to define the amount of key material that the attacker needs to recover in order to break the encryption.

```
1  #define ATTR(x) __attribute__((metamark(#x)))
2
3  void
4  toyCrypt (const uint32_t *_key,
5            const uint32_t *_input,
6            uint32_t *output) {
7
8    const uint32_t ATTR(key) *key = _key;
9    const uint32_t ATTR(plain) *input = _input;
10
11   const uint32_t subkeys[] = {
12                     key[0],
13                     key[1],
14                     key[0] ^ key[1],
15                     key[1] << 10 ^ 0xFCEF };
16
17 #define rotl(a,b) \
18   ((a) << (b)) |  \
19   ((a) >> ((sizeof(a) << 3) - (b)))
20 #define iter(n) rotl(state ^ subkeys[n], n+2)
21
22   uint32_t state = *input;
23   state = iter(0);
24   state = iter(1);
25   state = iter(2);
26   state = iter(3);
27   *output = state;
28 }
```

Figure 3.1: C code of the toyCipher

```
...
%sk0 = load i32* %key                    ; line 12

%ptr = getelementptr i32* %key, i64 1 ; line 13
%sk1 = load i32* %ptr

%sk2 = xor i32 %sk1, %sk0                ; line 14

%tmp = shl i32 %sk1, 10                  ; line 15
%sk3 = xor i32 %tmp, 64751

%state= load i32* %input                 ; line 22

%xor0 = xor  i32 %state, %sk0            ; line 23
%shl0 = shl  i32 %xor0, 2
%shr0 = lshr i32 %xor0, 30
%or0  = or   i32 %shl0, %shr0

%xor1 = xor  i32 %or0, %sk1              ; line 24
%shl1 = shl  i32 %xor1, 3
%shr1 = lshr i32 %xor1, 29
%or1  = or   i32 %shl1, %shr1

%xor2 = xor  i32 %sk2, %or1              ; line 25
%shl2 = shl  i32 %xor2, 4
%shr2 = lshr i32 %xor2, 28
%or2  = or   i32 %shl2, %shr2

%xor3 = xor  i32 %sk3, %or2              ; line 26
%shl3 = shl  i32 %xor3, 5
%shr3 = lshr i32 %xor3, 27
%or3  = or   i32 %shl3, %shr3

store i32 %or3, i32* %output             ; line 27
...
```

Figure 3.2: LLVM IR code: virtual registers in black, operations in blue, operand size in green

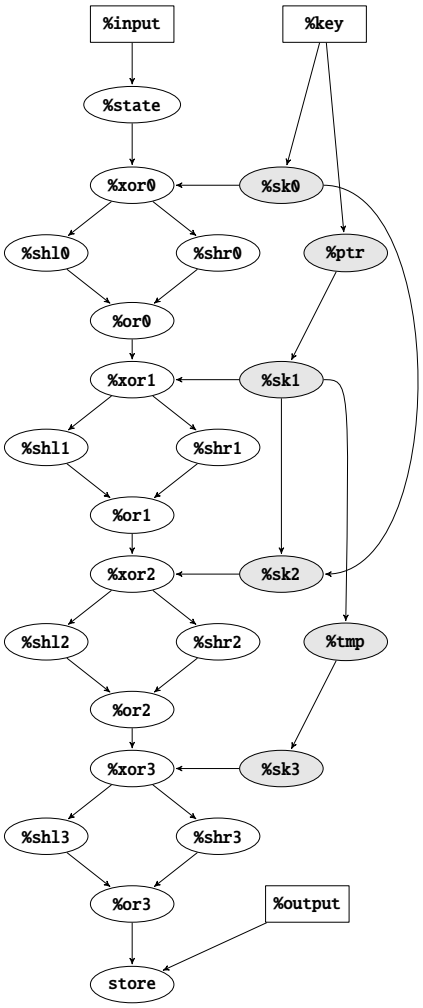


Figure 3.3: IR Data Dependency Graph.
Nodes belonging to the KEY-SCHEDULE are shown in gray

Definition 3.2.2 (Target key material). The target key material is a subset of the directly employed key material sufficiently large as to allow the attacker to reconstruct the encryption key.

In the most conservative scenario, the minimum amount of key schedule material necessary to reconstruct the encryption key is obtained when the

key schedule procedure is invertible. In this case, the target key material has the same bit size as the encryption key. The worst case scenario for the attacker is when needs to recover the entire key material.

Example 3.2.1. In the example in Section 3.2, the {`%sk0`,`%sk1`} {`%sk0`,`%sk2`} values are a valid choice for the target key material.

The most efficient choice for the attacker is thus to pick the values depending directly on a known value and a portion of the key material, possibly through a bitwise combination. In fact such a scenario allows a retrieval of the key material with a computational complexity which grows linearly with the number of key bits. The only way for an attacker to predict the result of an operation the inputs of which depend both on a known value and the key material is to consider all the possible values of the key bits involved in both values, thus increasing the difficulty of the attack. For instance, trying to make an hypothesis on the toy cipher state after a round has already been run (e.g. predict the value of `%xor1`) requires the attacker to make hypotheses on both the values of `%sk0` and `%sk1`. The attacker will therefore attempt to compute an *optimal choice* of the target key material, i.e. a sufficiently large set of key material bits which can be recovered with the minimum computational effort. In the example, the {`%sk0`, `%sk1`} is an optimal choice of target key materials, while {`%sk0`,`%sk2`} is not, as in this second choice the number of key bits that must be hypothesized to recover the value of `%sk2` is higher than the one for `%sk1`.

It is important to note that an alternate strategy for an attacker is available: instead of predicting the result of an operation combining the key with known inputs, it is possible for him to target in an attack the input of an operation involving the key, of which the output is known. The optimal choice of the target key material in this case will be different, as the attacker will analyze the cipher moving from the results back into the computation, but the same definitions and properties mentioned before can be re-stated in a straightforward manner.

Since the key material generated by the key schedule depends with some redundancy on the original encryption key, the attacker may be able to optimize the key retrieval simply hypothesizing the value of the (fewer) key bits on which the involved key material portion depends, further speeding up the attack.

3.2.1 Attack Surface Quantification Algorithm

This section describes the algorithm that uses the SDFA defined in 3.1 and in [1]. to detect the best intermediate value that can become the target of the hypothesis made by the attacker, among all of the possible ones. To the end of quantifying the attack surface, we now define the instruction resistance as:

Definition 3.2.3 (instruction resistance). The instruction resistance value represents the number of bits to be guessed with the attack choice most favorable to the attacker.

This is needed because the attacker might not choose to make hypothesis about the user key, because the key-scheduler mixed it in a very efficient way (e.g. in Serpent) and the hypothesis size might become too big to be computed. Another efficient way for the attacker is to make an hypothesis about the key material directly employed by the cipher, also called subkeys or output of the key scheduler. When the attacker has correctly guessed subkeys for at least the same amount of information contained in the user key, he has what he needs to invert the key scheduler and reconstruct the user key. If the key scheduler is not correctly designed, and does not mix every part of the key as soon as possible that operation is not possible, as the attacker would get a system of equations with redundant ones.The most intuitive case is the concatenation of multiple runs of the same algorithm, e.g. in Triple DES. From a designers' point of view, the risk is to over-protect the part of the key mixed sooner and leave unprotected the part of the key mixed later. This problem was dealt with through keeping track of every user key bit in the entire directly employed key material and choosing the vulnerable parts accordingly.
The operative subkey selection, shown in Algorithm 3.2.1, operates as follows: as a first operation, it performs a forward SDFA to compute the dependencies $M^{(\texttt{key})}$ of all the CFG nodes from the encryption key values (line 1). Subsequently it computes the subset $\mathcal{C}$ of nodes of the CFG which depend on the cipher input, i.e. the plaintext (lines 2–3). This set is the *core* of the cipher, and is the part potentially vulnerable to Differential Power Analysis attacks. This computation is performed initializing $\mathcal{C}$ to the nodes which define the plaintext, and iteratively adding to the set all the nodes

Algorithm 3.2.1: Cipher Attack Surface Quantification

Input: $\mathcal{G}=(\mathcal{V},\mathcal{E})$: IR data flow graph,
`key`: user encryption-key value,
$\mathcal{P}$: set of plaintext nodes ($\mathcal{P}\subset\mathcal{V}$),
$\mathcal{K}_{\texttt{key}}$: sequence of nodes that load the user cipher-key ($\mathcal{K}_{\texttt{key}}\in\mathcal{V}$)

Output: $\mathcal{C}\subset\mathcal{V}$: the set of nodes depending on the plaintext decorated with their resistance to power-based side-channel analysis

$\mathrm{M}^{(\texttt{key})} \leftarrow$ FORWARD-SDFA$(\mathcal{G},\mathcal{K}_{\texttt{key}})$

$\mathcal{C}_{\texttt{prev}}\leftarrow\emptyset$, $\mathcal{C}\leftarrow\mathcal{P}$

while $\mathcal{C}_{\texttt{prev}}\neq\mathcal{C}$ **do** $\mathcal{C}_{\texttt{prev}}\leftarrow\mathcal{C}$, $\mathcal{C}\leftarrow\mathcal{C}\cup\{v\in\mathcal{V}$ s.t. $\exists c\in\mathcal{C}_{\texttt{prev}},\ (c,v)\in\mathcal{E}\}$

$\mathcal{K}_{\texttt{prev}}\leftarrow\emptyset$, $\mathcal{K}\leftarrow\mathcal{K}_{\texttt{key}}$

while $\mathcal{K}_{\texttt{prev}}\neq\mathcal{K}$ **do** $\mathcal{K}_{\texttt{prev}}\leftarrow\mathcal{K}$,
$\mathcal{K}\leftarrow\mathcal{K}\cup\{v\in\mathcal{V}$ s.t. $\forall(v',v)\in\mathcal{E},\ v'\in\mathcal{K}_{\texttt{prev}}, v\notin\mathcal{C}\}$

$S_{\texttt{Fwd}}\leftarrow$ DETECTSUBKEYSBITS$(\mathcal{G},\mathrm{M}^{(\texttt{key})},\mathcal{P},\mathcal{K},\mathcal{C},\texttt{Fwd})$

$S_{\texttt{Bwd}}\leftarrow$ DETECTSUBKEYSBITS$(\mathcal{G},\mathrm{M}^{(\texttt{key})},\mathcal{P},\mathcal{K},\mathcal{C},\texttt{Bwd})$

$\mathrm{M}^{(\texttt{subkey-Fwd})}\leftarrow$ FORWARD-SDFA$(\mathcal{G},S_{\texttt{Fwd}})$

$\mathrm{M}^{(\texttt{subkey-Bwd})}\leftarrow$ BACKWARD-SDFA$(\mathcal{G},S_{\texttt{Bwd}})$

foreach $c\in\mathcal{C}$ **do**

  $\mathbf{m}_c\leftarrow\mathrm{M}^{(\texttt{key})}(c)$, $\overline{\mathbf{m}}_c\leftarrow\mathrm{M}^{(\texttt{subkey-Fwd})}(c)$, $\widehat{\mathbf{m}}_c\leftarrow\mathrm{M}^{(\texttt{subkey-Bwd})}(c)$
  // $\mathbf{m}_c$ is bitsize$(c)\times|\mathcal{K}_{\texttt{key}}|$
  // $\overline{\mathbf{m}}_c$ is bitsize$(c)\times(\sum_{\texttt{s}}\texttt{bitsize}(s),s\in S_{\texttt{Fwd}})$, $\widehat{\mathbf{m}}_c$ is bitsize$(c)\times(\sum_{\texttt{s}}\texttt{bitsize}(s),s\in S_{\texttt{Bwd}})$

  min $\leftarrow+\infty$ // BITLEN(key) is also a correct initialization value

  for idx $\leftarrow 0$ **to** BITLEN$(c)-1$ **do**

    if min $>$ HW$(\mathbf{m}_c[\text{idx},:])$ AND HW$(\mathbf{m}_c[\text{idx},:])>0$ **then**
    min $\leftarrow$ HW$(\mathbf{m}_c[\text{idx},:])$

    cnt $\leftarrow 0$, vect_Fwd $\leftarrow(0,\ldots,0)$ // length(vect_Fwd)=bitsize(key)
    for $j\leftarrow 0$ **to** $|S_{\texttt{Fwd}}|-1$ **do**
      $f\leftarrow S_{\texttt{Fwd}}[j]$, $\mathbf{m}_f\leftarrow\mathrm{M}^{(\texttt{key})}(f)$
      for $h\leftarrow 0$ **to** bitsize$(f)-1$ **do**
        if $\overline{\mathbf{m}}_c[\text{idx},\texttt{cnt}]=1$ **then** vect_Fwd $\leftarrow$ vect_Fwd $\vee\ \mathbf{m}_f[\texttt{cnt},:]$
        cnt $\leftarrow$ cnt $+1$
    if min $>$ HW(vect_Fwd) AND HW(vect_Fwd) >0 **then**
    min $\leftarrow$ HW(vect_Fwd)
    if min $>$ HW$(\overline{\mathbf{m}}_c[\text{idx},:])$ AND HW$(\overline{\mathbf{m}}_c[\text{idx},:])>0$ **then**
    min $\leftarrow$ HW$(\overline{\mathbf{m}}_c[\text{idx},:])$

    cnt $\leftarrow 0$, vect_Bwd $\leftarrow(0,\ldots,0)$ // length(vect_Bwd)=bitsize(key)
    for $j\leftarrow 0$ **to** $|S_{\texttt{Bwd}}|-1$ **do**
      $b\leftarrow S_{\texttt{Bwd}}[j]$, $\mathbf{m}_b\leftarrow\mathrm{M}^{(\texttt{key})}(b)$
      for $h\leftarrow 0$ **to** bitsize$(b)-1$ **do**
        if $\widehat{\mathbf{m}}_c[\text{idx},\texttt{cnt}]=1$ **then** vect_Bwd $\leftarrow$ vect_Bwd $\vee\ \mathbf{m}_b[\texttt{cnt},:]$
        cnt $\leftarrow$ cnt $+1$
    if min $>$ HW(vect_Bwd) AND HW(vect_Bwd) >0 **then**
    min $\leftarrow$ HW(vect_Bwd)
    if min $>$ HW$(\widehat{\mathbf{m}}_c[\text{idx},:])$ AND HW$(\widehat{\mathbf{m}}_c[\text{idx},:])>0$ **then**
    min $\leftarrow$ HW$(\widehat{\mathbf{m}}_c[\text{idx},:])$

  c.resistance $\leftarrow$ min

return $\mathcal{C}$

which follow them in the CFG, until the set $\mathcal{C}$ is stable. Once $\mathcal{C}$ has been computed, it is possible to derive which nodes of the CFG are computing values dependent only on the encryption key. The set of these nodes, $\mathcal{K}$, is computed at lines (line 4–5), through initializing it with the definition nodes of the encryption key $\mathcal{K}_{\texttt{key}}$, and adding iteratively the nodes having all the ancestors already in $\mathcal{K}$. Those nodes should not be present in $\mathcal{C}$ as otherwise they would depend also on the plaintext. The set $\mathcal{K}$ represents the portion of the algorithm computing the key schedule of the block cipher. After computing $\mathcal{C}$ and $\mathcal{K}$ the algorithm proceeds to compute which nodes of the directly employed key material nodes, and, consequently, the optimal target key material. This computation, performed by the DETECTSUBKEYSBITS primitive described in the next section, is carried out taking into account both the attacks which can be led knowing the input to the algorithm, and guessing values derived from it by the computation, and the ones lead knowing the output of the algorithm and inferring intermediate values generating it. The first analysis is called *forward optimal target key material search* and the latter is the *backward* one (as denoted in lines 6–7). The computed sets denoted respectively as $S_{\texttt{Fwd}}$ and $S_{\texttt{Bwd}}$ contain the smallest set (respectively, at the top or at the bottom of the block cipher) of target key material nodes that depend from the entire encryption key. Once these two analysis steps have been completed, the algorithm performs a forward SDFA computing the data dependencies $M^{\texttt{subkey-Fwd}}$ of the output values from the forward optimal target key material $S_{\texttt{Fwd}}$ (line 8) and, acts similarly performing a backward SDFA for the backward optimal target key material $S_{\texttt{Bwd}}$. Once the dependencies from the encryption key, forward and backward optimal target key materials have been computed, the algorithm proceeds to quantify the resistance level of each instruction (line 10). For each instruction $c \in \mathcal{C}$ of the cipher, the algorithm works as follows. As a first step, the algorithm stores the dependencies of the current instruction c from the encryption key, forward and backward optimal target key materials respectively into the leakage vectors $\mathbf{m}_c, \overline{\mathbf{m}}_c$ and $\widehat{\mathbf{m}}_c$ (line 11). The temporary variable `min` stores the candidate value for the instruction resistance of c and it is initialized to $+\infty$ (or equivalently to BITLEN(`key`)) (line 12). Since the analysis is done separately for each bit of each instruction, they are computed separately. So for each output bit of each visited node (line

13-30), there are three *SDFA* results available:

1. **The dependencies from the user encryption key**: represented by the column of the matrix $\mathbf{m}_c$ containing the dependencies of the current bit from the user key material as determined by Forward SDFA. (line 14)

2. **The dependencies from the *most vulnerable* set of subkeys from the top of the algorithm**, determined by Forward SDFA and stored in the column of matrix $\overline{\mathbf{m}}_c$ representing this bit. (line 22)

3. **The dependencies from the *most vulnerable* set of subkeys from the bottom of the algorithm**, determined by Backward SDFA and stored in the column of matrix $\widehat{\mathbf{m}}_c$ representing this bit. (line 30)

If some group of instructions in sets $S_{\texttt{Fwd}}$ or $S_{\texttt{Bwd}}$ depends on less user key bits than their own size, the attacker can try to make hypothesis on this smaller subset of userkey bits and reconduct the attack as it was a normal attack using $S_{\texttt{Fwd}}$ or $S_{\texttt{Bwd}}$ as keys. To consider this possibility, the dependencies from the directly employed key material are expressed as dependencies from the userkey (lines 18-21 and 26-29) and this generates other two attack possibilities. The instruction resistance value represents the number of bits to be guessed with the attack choice most favorable to the attacker (line 31). To compute the instruction resistance value, the first step is to obtain the instruction resistance value for each of the five attack possibilities. They are obtained by computing the Hamming Weights (HW) for each row of the corresponding matrix and picking the lowest one, save from the cases when it's zero. Computing the minimum among those five values, yields the effective instruction resistance.

Example 3.2.2 (Running example). Starting from the node `%_input`, the procedure marks the portion of the DFG depending on the at least a node depending on the plaintext as part of set $\mathcal{C}$ (lines 2–3 of Algorithm 3.2.1). In Figure 3.3, these nodes are reported as ovals with white background. Then, starting from the node `%_key`, the procedure mark the nodes of the DFG depending on the key and not on the plaintext as part of set $\mathcal{K}$ (lines 4–5). In Figure 3.3, these nodes are reported as ovals with a shaded background.

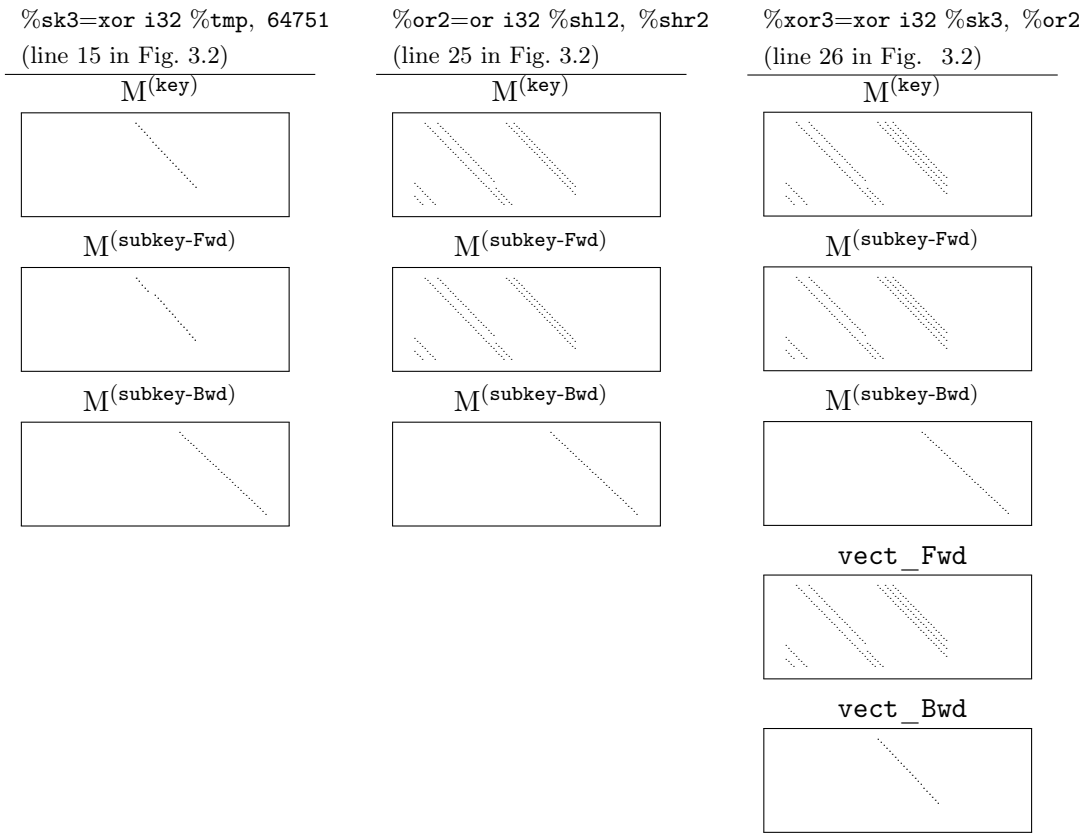


Table 3.1: Leakage vectors for the toyCipher example, instruction %xor24 and key material nodes.

Using procedure DetectSubkeyBits, the algorithm obtains the optimal target key material nodes for both the forward and the backward attack, and computes the corresponding leakage vectors by propagating their effects using the SDFA (lines 6–9).

In Figure 3.3, the optimal target key material nodes are $S_{\texttt{Fwd}} = \{\%\texttt{sk0}, \%\texttt{sk1}\}$ and $S_{\texttt{Bwd}} = \{\%\texttt{sk2}, \%\texttt{sk1}, \%\texttt{sk3}\}$.

Then the algorithm considers each node in $\mathcal{C}$ to compute its resistance. Considering as an example the instruction %xor24, Table 3.1 reports the leakage vectors computed for it and corresponding to the four possible attacks: optimal target key material forward ($\mathrm{M}^{(\texttt{subkey-Fwd})}$) and backward ($\mathrm{M}^{(\texttt{subkey-Bwd})}$), and encryption key forward (vect_Fwd) and backward (vect_Bwd). Also reported is the leakage vector $\mathrm{M}^{(\texttt{key})}$ obtained at line 1 from the initial forward SDFA on the encryption key, which is used to perform the reverse

lookup on the original key- subkey bit dependencies for all the instruction and find the actual resistance. The matrices represented in table 3.1 are the ones obtained at the end of the algorithm, have the key bits on the X axis and the data bits of the corresponding SSA register on the Y axis. Having a black dot at the intersection, means that the data bit depends on the corresponding key bit, according to the respective data flow analysis.
The resistance of %`xor24` is then easily obtained by taking the minimum Hamming Weight for the four leakage vectors. The inspection of `vect_Bwd` makes it easy to understand that the instruction resistance of %`xor24` is 1, as all rows of the leakage vector have the same, unitary, Hamming Weight.

3.2.2 Optimal Target Key Material Search

We now detail how the DetectSubkeysBits finds which are the optimal target key materials employed in our analysis. The algorithm is applied two times during an analysis of a block cipher, as this analysis can be performed in two directions. We will now provide a high level description of the algorithm, for clarity's sake, followed by a detailed analysis of the steps.
A forward analysis detects the directly employed key material which is used as input of a CFG node as soon as possible, while a backward one will target the directly employed key material used as late as possible. The algorithm operates on a sorted (by distance from plaintext or ciphertext) list of potentially vulnerable instructions. It keeps track of user key bits already *covered* by selected instructions, and each instruction can cover (or it can give to an attacker informations about) a number of bits at maximum equal to the size in bits of its SSA register. By iterating on the list, the algorithm selects instructions that cover at least one bit and adds them to the list of *effectively vulnerable* ones, until the entire user key is covered by the smallest possible set of instructions. The algorithm encounters situations where an instruction depends on more user key bits (still not covered) than its size and has to make a choice. Those choices are done greedily, and added to a decision list which is re-examined (backtracking) at each new instruction, until the doubt is not relevant anymore. This guarantees an optimal result. The problem is further complicated by the fact that two or more instructions that have the same distance (measured in hops of definition/uses) from the plaintext are equally interesting for an attacker and

should be considered equally dangerous. This is worked out by grouping the sorted list for equal values of distances, counting uncovered bits appropriately and deciding which instructions are dangerous considering this fact.

Algorithm 3.2.2: Detection of Subkeys

Input: $\mathcal{G}{=}(\mathcal{V},\mathcal{E})$: IR data flow graph
$\mathcal{P}$: set of plaintext nodes ($\mathcal{P} \subset \mathcal{V}$),
$\mathcal{K}$: set of nodes which define key material,
$\mathcal{C}$: set of nodes depending on the ones in $\mathcal{P}$,
$\mathrm{M}^{(\mathtt{key})}$: Map of user key dependencies for every element of $\mathcal{V}$,
$\mathtt{dir} \in \{\mathtt{Fwd}, \mathtt{Bwd}\}$: parameter denoting the direction of the analysis

Output: S

Data: $\mathcal{D}$: Set of doubts in subkey bit choices

$\mathcal{K}_{\mathtt{out}} \leftarrow \{k{\in}\mathcal{K} \text{ s.t. } \exists\, c{\in}\mathcal{C},\ (k,c){\in}\mathcal{E}\}$
$\mathcal{C}_{\mathtt{out}} \leftarrow \{c{\in}\mathcal{C} \text{ s.t. } \nexists\, d{\in}\mathcal{C}, (c,d){\in}\mathcal{E}\}$
foreach $k{\in}\mathcal{K}_{\mathtt{out}}$ **do**
  $\mathcal{U}_k \leftarrow \{c{\in}\mathcal{C} \text{ s.t. } (k,c){\in}\mathcal{E}\}$
  if $\mathtt{dir} = \mathtt{Fwd}$ **then** $k.\mathtt{dist} \leftarrow |\textsc{shortestPath}(\mathcal{P}, \mathcal{U}_k)|$
  else $k.\mathtt{dist} \leftarrow |\textsc{shortestPath}(\mathcal{U}_k, \mathcal{C}_{\mathtt{out}})|$
$\mathtt{cov\_uk} \leftarrow (0,\ldots,0)$ `/* |cov_uk| = |key|; tracks the user key bits on which the subset of chosen subkey bits depend */`
$\mathcal{D} \leftarrow \varnothing$, $\mathcal{S} \leftarrow \varnothing$
for $i \leftarrow 1$ **to** $\max(\{k.\mathtt{dist} | k \in \mathcal{K}_{\mathtt{out}}\})$ **do**
  $\mathcal{L} \leftarrow \{k \in \mathcal{K}_{\mathtt{out}} | k.\mathtt{dist} = i\}$
  $\mathtt{cov\_L} \leftarrow (0,\ldots,0)$ `/* |cov_L| = |key| */`
  foreach $l \in \mathcal{L}$ **do**
    $\mathbf{m}_l \leftarrow \mathrm{M}^{(\mathtt{key})}[l]$, $\mathtt{uk\_deps} \leftarrow (0,\ldots,0)$ `/* |uk_deps| = |key| */`
    for $\mathtt{bit} \leftarrow 0$ **to** $\textsc{size}(l) - 1$ **do** $\mathtt{uk\_deps} \leftarrow \mathtt{uk\_deps} \vee \mathbf{m}_l[\mathtt{bit}, :]$
    $\mathtt{cov\_l} \leftarrow \mathtt{uk\_deps} \wedge (\neg \mathtt{cov\_uk})$
    $\mathcal{D} \leftarrow \mathcal{D} \setminus \{d \in \mathcal{D} | \mathrm{HW}(d.left \wedge \mathtt{cov\_uk} \wedge \mathtt{cov\_L}) = 0\}$
    $(\mathtt{mand}, \mathtt{cov\_uk}, \mathcal{D}) \leftarrow \textsc{CheckForMandatories}(\mathtt{cov\_uk}, l, \mathcal{D}, \mathtt{uk\_deps}, \mathtt{cov\_l})$
    $\mathtt{cov\_l} \leftarrow \mathtt{cov\_l} \vee \mathtt{mand}$ `/* |cov_l| = |key| */`
    if $\mathrm{HW}(\mathtt{cov\_l}) > 0$ **then**
      $\mathtt{cov\_l} \leftarrow \mathtt{cov\_l} \wedge (\neg \mathtt{cov\_L})$
      $(\mathtt{cov\_l}, \mathcal{D}) \leftarrow \textsc{LimitTakenBits}(size(l), \mathtt{cov\_l}, \mathtt{mand}, \mathcal{D})$
      $\mathtt{cov\_L} \leftarrow \mathtt{cov\_L} \vee \mathtt{cov\_l}$
      $\mathcal{S} \leftarrow \mathcal{S} \cup \{l\}$
  $\mathtt{cov\_uk} \leftarrow \mathtt{cov\_uk} \vee \mathtt{cov\_L}$
  if $\mathtt{cov\_uk} = (1,\ldots,1)$ **then break**
return S

Here follows a detailed description of Algorithm 3.2.2: As a first action, the algorithm identifies the directly employed key material $\mathcal{K}_{\texttt{out}}$, through finding the nodes in the set of key schedule nodes $\mathcal{K}$ which have an outgoing edge in the set of cipher nodes $\mathcal{C}$ (line 1). This is followed by the identification of the nodes of the output values of the cipher (set $\mathcal{C}_{\texttt{out}}$), as the nodes in $\mathcal{C}$ which have no outgoing edges (line 2). Once these two sets have been obtained, the algorithm computes for each directly employed key material value k the set of nodes $\mathcal{U}_k$ representing an use of k, and the minimum distance between any node in $\mathcal{U}_k$ and either the plaintext or the output values, depending on the analysis direction set by the parameter `dir` (lines 3–6).

Having obtained a distance value for all the nodes representing the result of an use of the directly employed key material, the algorithm starts building the set of nodes containing the optimal target key material $\mathcal{S}$, keeping track of the encryption key bits which can be recovered from the current $\mathcal{S}$ in `cov_uk` (lines 7–8). The binary vector `cov_uk` has one bit for each bit of the encryption key. Each bit of `cov_uk` indicates whether the corresponding bit of the encryption key contributes to the computation of an output value in $\mathcal{S}$. To this end we require that the nodes in $\mathcal{S}$ depend on at least as many directly employed key material bits as the ones of the encryption key, and that all the encryption key bits are represented, i.e., the binary vector `cov_uk` is filled with ones. As this procedure is completed in a single sweep over the nodes in $\mathcal{K}_{\texttt{out}}$, it is possible that, upon the need of choosing which encryption key bits are represented by the outputs of $k \in \mathcal{K}_{\texttt{out}}$, some degree of freedom is allowed in the choice. This is a consequence of the output bits of k depending on more encryption key bits than their effective number. We record as an element of the decisions set $\mathcal{D}$ which encryption key bits could have been considered covered by an instruction $k \in \mathcal{K}_{\texttt{out}}$, and were not due to the lack of output material. The set of decisions $\mathcal{D}$ is used to perform optimal backtracking of the choices in the CheckForMandatories function, should the need arise.

After having initialized the set of decisions to the empty set, the algorithm starts iterating over the nodes of $\mathcal{K}_{\texttt{out}}$, dividing them in layers according to their distance value (lines 9–11). In particular, the current layer is stored in the set $\mathcal{L}$, and an auxiliary variable tracking the encryption key bits

covered by the nodes picked in the layer `cov_L` is initialized. For each node l of the layer $\mathcal{L}$, an auxiliary variable tracking its dependencies from the encryption key bits `uk_deps` is initialized, and the matrix $\mathbf{m}_l$ of all its bitwise dependencies from them is extracted. (lines 12–13). The dependencies in $\mathbf{m}_l$ are accumulated via bitwise or into `uk_deps` (line 14). The `cov_l` variable, indicating which encryption key bits are represented by the current node l, is initialized to the ones in `uk_deps` minus the ones which are already represented by variables in previous layers. The already represented bits are held in `cov_uk` (line 15). The choice of encryption key bits may be suboptimal, if performed on the basis of local information. In particular, it is possible for a encryption key bit to be represented only by a single directly employed key material bit in the algorithm. Consequentially, a choice based on a local criterion may exclude it in a situation where not enough output bits are available, resulting in a potentially incomplete optimal target key material selection. To prevent this from happening, the algorithm employs a call to the CheckForMandatories function, which detects such cases through picking the optimal choice of key bits employing the information contained in the decision set $\mathcal{D}$ to perform eventual backtracking. To this end, first step is to perform basic housekeeping on the $\mathcal{D}$ set, removing all the leftover decisions involving key bits which have been covered in the current layer $\texttt{cov}_\texttt{l}$, or by the currently selected portion of the optimal target key material `cov_uk` (line 16). Subsequently, the CheckForMandatories routine is invoked, and it returns the set of the encryption key bits which are mandatory to be represented by the current instruction `mand`, together with an updated set of decisions $\mathcal{D}$ and a possibly updated value for `cov_uk`. The bits in `mand` are mandatory because they were considered as covered before, and the enhancement of the previous decision requires those to be still covered as soon as possible.

The set of encryption key bits which must be represented by the output of the current instruction are added to the selection $\texttt{cov}_\texttt{l}$ (line 18), and it is checked whether the current value of $\texttt{cov}_\texttt{l}$ includes at least a bit (i.e. its Hamming Weight is greater than zero) (line 19). If this is the case, the output of the current instruction is considered tentatively as a part of the optimal target key material, and thus it is evaluated which encryption key bits are represented by it. To this end, the encryption key bits already cov-

ered by other instructions in the same layer are removed (line 20), and the function LimitTakenBits checks that there is enough output material in the instruction l to represent all the key bits. If this is not the case, it selects a subset of them, adding a new element to the decision set $\mathcal{D}$, to track the ones left out. Note that the LimitTakenBits function takes as input the mandatory encryption key bits to be covered by the current instruction and aborts the algorithm execution in case it is not possible for the current instruction to cover them. After the number of encryption key bits covered by l is properly limited, they are added to the ones covered by the current layer (line 22), and the instruction is added to the set $\mathcal{S}$ (line 24). Each time an entire layer is processed, the algorithm merges the bits covered by the layer with the ones in `cov_uk` updating the current encryption key coverage of the optimal target key material, halting as soon as the entire encryption key is fully covered (line 25). In case the algorithm terminates without having reached this condition, the encryption key bits are not fully employed in the algorithm, pointing to a significant cipher flaw. It is possible for the designer to inspect the results to the end of correcting the cipher flaw, as

the algorithm provides precisely which encryption key bit are neglected.

Algorithm 3.2.3: CheckForMandatories

Input: `cov_uk`: bits covered in previous layers
l: the current instruction l
$\mathcal{D}$: Set of decisions made in subkey bit choices: models the set of choices made with a limited forward knowledge of the DFG
Each $d \in \mathcal{D}$ is $d = (\texttt{taken}, \texttt{left})$
Output: `mand`: bits which must be covered by instruction l
`cov_uk`: updated bits covered in previous layers
$\mathcal{D}$: Set of updated doubts in subkey bit choices

```
1  mand ← (0, ..., 0)
2  foreach d ∈ D do
       /* If d has previously left bits which l cannot cover          */
3      if HW(d.left ∧ (¬cov_l)) > 0 then
           /* And some which were covered can be covered by l         */
4          adoptable ← d.taken ∧ uk_deps
5          if HW(adoptable) > 0 then
6              quantity_to_swap ← size(l)−HW(mand)
7              mand_new ←GetFirstNBits(adoptable, quantity_to_swap)
8              mand ← mand ∨ mand_new
9              d.taken ← d.taken ∧ (¬mand_new)
10             cov_uk ← cov_uk ∧ (¬mand_new)
11             sent_back ←GetFirstNBits((d.left ∧ (¬cov_l)),HW(mand_new))
12             d.left ← d.left ∧ (¬sent_back)
13             d.taken ← d.taken ∨ sent_back
14             cov_uk ← cov_uk ∨ sent_back
15             if HW(mand) = size(l) then break
   return ⟨mand, cov_uk, D⟩
```

We now detail the behavior of the CheckForMandatories function, employed by the DetectSubkeysBits as reported in Algorithm 3.2.3. The algorithm iterates over all the decisions d in the decision set $\mathcal{D}$ (line 2) and checks whether a previous decision could have been taken in such a way that its discarded bits can be represented by l, and the ones which l is not able to represent can be in turn covered revising the previous decision (lines 3–5). If this choice is viable, the decision is revised so that the previously discarded bits are effectively mandated to be covered by l, while in turn they are removed from the ones being in need to be covered. Note that there is no effective need to change the previous bit assignments as the whole instruction output is marked to be part of the optimal target key material, as an attacker gaining information on a subset of its output bits is able to

obtain it on all of them, since the measurements he has acquired contains all the informations.

3.3 Automated Masking Against Passive SCA

This section describes the masking technique used in this work and how it was automatically applied to the relevant portions of the block ciphers.
Masking aims at invalidating the link between the predicted power consumption, associated to an intermediate operation, and the corresponding power measurement. In a masked implementation, each sensitive intermediate value is concealed through splitting it in a number d+1 of shares, which are separately processed [13,17]. The unprotected computation is substituted by three phases: an initial share-splitting, a transformation of the original computation into one processing all the d+1 shares and a final recombination, which must yield the same result as the unprotected computation. A number of masking schemes have been proposed in the literature. In this work we will employ the one by Ishai *et al.* [13].
Table 3.2 shows the computational costs to mask bitwise operations as a function of the scheme order d. The computational costs to perform a d-th order masked bitwise operation are explicited in the following paragraph. A masked `xor` operation is computed through $2d(d+1)$ `xor` operations, while the `not` operation has the same cost of an unprotected one. A masked `and` operation takes $2d^2+d+1$ `xor` and d^2+d+2 `and` operations to be computed. It is possible to express an `or` operation through converting it in ANF as $a \vee b = a \oplus b \oplus ab$. As ciphers often involve the computation of multi-bit nonlinear Boolean functions, these can either be expressed in Algebraic Normal Form (ANF) and explicitly computed, or be available as a lookup table. In the case of tabulated nonlinear functions, masked table lookup operations on a l-element lookup table costs $2ld$ `xor` operations, ld `store` operations, and ld+1 `load` operations. For multi-bit arithmetic operations it is possible to perform conversions between Boolean masked values and arithmetic masked ones and vice versa [10]. The masking scheme employed in this work for generic Boolean functions is proven to be secure up to a d-th order attack [13] for any choice of $d \geq 1$. In case the Boolean function is available in the form of a lookup table, the masking of the looked-up values is safe

Table 3.2: Complexity of bitwise masked operations as a function of the masking order d and lookup table size l

Operation	Complexity of masked operation	Ref.
`xor`	$3(d{+}1)$ `xor`	[13, 22]
`not`	1 `not`	[13]
`and`	$2d(d{+}1)$ `xor` $+(d{+}1)^2$ `and`	[13]
`or`	$2d(d{+}1)$ `xor` $+(d{+}1)^2$ `and` $+3$ `not`	$a \vee b = \neg((\neg a) \wedge (\neg b))$
table lookup	$2ld$ `xor` $+ld$ `store` $+(ld+1)$ `load`	[23]

up to the 2nd order according to [9]. The key idea is that, whenever two share-split operands are combined together, fresh random values should be inserted in the computation of the resulting output shares. As the masking countermeasure is particularly computationally demanding (see Table 3.2) applying it as sparingly as possible, without lowering the security margin of the cipher, is paramount.

3.3.1 Automated Masking Application

The mask application compiler pass uses the previous masking schemes, employing the information provided by the Attack Surface Quantification Algorithm 3.2.1 to select which instructions should be protected. The masking application pass visits the CFG in order, and, for each instruction, acts as follows. In case the instruction output needs to be protected, it checks whether the instruction operands are available in masked form or not. In the former case, it simply emits the masked operation corresponding to the unprotected one under exam. In the latter case, it first emits the code required to split the instruction operands into $d{+}1$ shares, followed by the masked operation code. In case the output of the instruction being visited does not need to be protected, the compiler pass checks whether its operands are masked or not. In case the operands are masked, the pass inserts the share recombination step to obtain an unmasked value. Subsequently, all the uses of the output of the instruction under exam are updated so that they employ the result of the share recombination. In case the operands of the instruction are already unprotected, the pass continues the visit of the CFG. Throughout this procedure, the only operations being computed in masked form are the ones in need to be protected, and the mask insertion and re-

moval phases are automatically applied right before and after the protected computation. After the protected operations have been added to the CFG, their unprotected correspondents are evicted, checking, as a safety measure, that no data dependencies have been broken in the process. Performing the mask application pass could benefit from a customized Boolean optimization pass which transforms the Boolean expressions of the cipher in ANF.

3.3.2 Masking Application to Single Instructions

Masked values

The share-splitting phase of a d+1-shares $s_0 \dots s_d$ masked algorithm (also known as a d-th order masked implementation) splits an input value i into d+1 shares employing d randomly chosen values $r_0 \dots r_{d-1}$ through computing $s_0{=}((i \oplus r_0) \oplus r_1) \oplus \dots) \oplus r_{d-1}$, and $s_i{=}r_{i-1}$ for $i{\in}\{1, \dots, d\}$. Note that, for the security of the scheme to hold, two random values must never be combined directly through a single operation. The final share recombination phase will exploit the property of the share splitting scheme to obtain the unmasked output value o as recombining its shares o_j as $\bigoplus_{j=0}^{d} o_j$. A masked-`xor` operation is thus computed through $2d(d{+}1)$ `xor` operations. To transform a typical block cipher into its version computing on d+1 shares, its primitives are classified in linear (over $(\mathbb{Z}_2, \oplus, \wedge)$) and nonlinear ones.

Linear operations

All linear primitives can be decomposed in a sequence of `xor` and `not` operations (also denoted as $\oplus$ and $\neg$, respectively), which act on the shares. The `not` operation acting securely on the d+1 shares is simply defined as $(\neg s_0)s_1 \dots s_d$ (as its computation does not yield any side channel information). Consequently, a masked-`not` operation has the same computational cost of an unmasked one. The $\oplus$ operation combining two values $u_0 \dots u_d$ and $v_0 \dots v_d$ into $o_0 \dots o_d$ needs to employ a set of d *fresh* random values $\{r_0, \dots, r_{d-1}\}$ in order for the security of the scheme to hold, and computes the values $o_i{=}\widetilde{u}_i \oplus \widetilde{v}_i$, where $\widetilde{u}_i{=}u_i \bigoplus_{j=0}^{d-1} r_j$ and $\widetilde{v}_i{=}v_i \bigoplus_{j=0}^{d-1} r_j$.

Non linear functions

The nonlinear primitives may either be implemented as their straightforward computation [13], or stored as a lookup table [23]. In case the nonlinear primitives are explicitly computed, the current state-of-the-art provably-secure masking scheme defines how to tackle the problem of computing a masked bitwise $\wedge$ operation. As every Boolean function can be expressed as a sequence of `and` ($\wedge$) and `not` operations, there are no restrictions on the form of the nonlinear function. Synthetically, the algorithm is: To compute

Input: $\langle \mathtt{a}_1, \mathtt{a}_2, \ldots, \mathtt{a}_n \rangle$: n shares representing the first operand (a)
$\langle \mathtt{b}_1, \mathtt{b}_2, \ldots, \mathtt{b}_n \rangle$: n shares representing the second operand (b)
Output: $\langle \mathtt{c}_1, \mathtt{c}_2, \ldots, \mathtt{c}_n \rangle$: n shares representing the result of the computation $a \wedge b$

```
1  begin
2      for i ← 1 to n do
3          for j ← 1 to n do
4              if i < j then
5                  z[i][j] ← Random()
6              else if i > j then
7                  x ← z[j][i] ⊕ (a_j ∧ b_i)
8                  z[i][j] ← x ⊕ (a_i ∧ b_j)
9      for k ← 1 to n do
10         t ← 0
11         for l ← 1 to n do
12             if k! = l then
13                 t ← t ⊕ z[k][l]
14         c_k ← (a_i ∧ b_i) ⊕ t
15     return ⟨c_1, c_2, ..., c_n⟩
```

$u_0 \ldots u_d \wedge v_0 \ldots v_d$, the first step is to obtain $\frac{d(d+1)}{2}$ *fresh* random values and assign them to $z_{i,j}$ for $i, j \in \{0, \ldots, d\}$, with $i<j$. Subsequently, the intermediate values $z_{j,i}$ are computed as $z_{j,i}=(u_i \wedge v_j) \oplus z_{i,j} \oplus (u_j \wedge v_i)$. Finally, the output values $o_0 \ldots o_d$ are obtained as $o_i=(a_i \wedge b_i) \bigoplus_{i \neq j} z_{i,j}$. Note that the final sum yielding o_i must be computed taking care of not composing two values of $z_{i,j}$ directly among them. On the overall, a masked $\wedge$ operation takes $2d^2+d+1$ `xor` and d^2+d+2 `and` operations to be computed.

S-BOXes

In case the nonlinear primitive is available as a lookup table `Sbox`[·], the masked computation of it exploits directly the definition of masking scheme as share splitting, and involves recomputing the entire lookup table, taking into account the effect of the random values. To this end a fresh set of d random values $\{r'_0, \ldots, r'_{d-1}\}$ is extracted and, for all the possible input values i of `Sbox`[·], the value $\texttt{Sbox}_{\text{masked}}[i \bigoplus_{j=1}^{d} s_j] = \texttt{Sbox}[i] \bigoplus_{j=0}^{d-1} r'_j$ is computed. After the computation of $\texttt{Sbox}_{\text{masked}}[\cdot]$, the output of the secure computation of `Sbox`[·] is defined as: $o_0 = \texttt{Sbox}_{\text{masked}}[s_0]$, and for all $i \in \{1, \ldots, d\}$, $o_i = r'_{i-1}$. The total cost of performing a masked table lookup operation on a l-element lookup table is of $2ld$ `xor` operations, ld `store` operations, and $ld+1$ `load` operations.

The fully computational scheme is proven to be secure up to a d-th order attack [13], while the lookup table re-computation technique is safe only up to the 2nd order as noted in [23]. Note that, if there is the need to perform `xor` operations while computing the nonlinear primitives of the cipher, it is possible to avoid to expand the `xor` into its `and`/`not` representation, employing the same masking technique used for masking of linear operations. Note that, although the fully computational method is able to mask any Boolean function, in case of ciphers with multi-bit arithmetic operations it is possible to employ more efficient arithmetic-masking techniques, or to perform conversions between Boolean masked values and arithmetic masked ones [23].

3.4 Implementation Techniques

The first step needed to be able to implement those algorithms in the LLVM framework, is to have a way to mark the key and the plaintext in the source code of the cipher. For this reason, the `clang` front-end was expanded to recognize a custom attribute to local variables declaration. This attribute is called *metamark*, and accepts a string parameter which is not checked by the compiler but only by further stages. A decorated source code looks like:

The front-end will produce regular `alloca` instructions, with LLVM IR Metadata of type MetaMark carrying the string as a parameter. Instead of

```
void block_cipher(uint8_t* _key,uint8_t* _plaintext,
        uint8_t* ciphertext){
    uint8_t* key __attribute__((metamark("chiave"))) = _key;
    uint8_t* plaintext __attribute__((metamark("plain")))
                                                = _plaintext;
    // Regular unrolled code of the block cipher
    // using key and plaintext variables.
}
```

Figure 3.4: Example of decorated source code

fine tuning the pipeline scheduler of LLVM for our needs, for development purposed the passes were executed in the required order using consecutive calls to the `opt` tool, which executed one or more steps on an input IR and outputs the optimized IR.
For analysis purposes, the entire program should be in SSA form, without `load`/`store` IR instructions, except for the ones at the beginning and at the end of the algorithm concerning input/output. However, as the mark produced by the front-end is applied only to the `alloca` instructions, an optimization pass was implemented to recursively copy the MetaMark attributes on every uses of memory allocated by marked `alloca` instruction. Being recursive means that every use of those uses is also marked, until the use set to mark becomes empty. This is the first pass applied to the IR code produced by the front-end.

After the MetaMark propagation pass is run,the LLVM framework Scalar Replacement of Aggregates (SROA) pass is run to decompose the single cells of arrays and matrices allocated on the stack of a function with an `alloca` into single SSA registers. After that, it's still possible to find some uses of memory operations in the code, which is still not optimized. The `-O3` set of optimizations gets rid of them in order to optimize the code, as optimizations, especially the most aggressive ones, are easier to write in pure SSA form.
The `InstructionCombine` pass, contained into the `-O3` set, among the other optimizations, recognizes if a `xor` could be safely replaced by an `or`. This is a recognizable condition, especially when there are fixed $\wedge$ masks on both operands, and is a useful micro-optimization: in some architectures it's more efficient to do a $\vee$ rather than a $\oplus$. However the masking of a $\vee$ instruction is the least efficient, and the masking of a $\oplus$ is the most efficient one between all binary operators. The `InstructionCombine` pass was patched to eliminate this counterproductive (for our needs) optimization. So the next step is the

`-O3` optimization set. After that step, many instructions were replaced so their MetaMark was lost. In order to restore it, another run of the pass which propagates them is able to fill in the blanks in the use chains of initial variables. The code is now ready to be analyzed by the proposed algorithm. For each instruction, the pass keeps a significant amount of data about them, as listed in Table 3.3.

Table 3.3: List of the most relevant informations kept for each instruction

bool isAKeyOperation;	True if the instruction value depends on the key.
bool isAKeyStart;	True if this instruction is the first one that produces the key value, so reads the key from memory
bool isVulnerableTopSubKey;	True if this instruction is marked as a value produced by the key scheduler, within the first ones to include all of the user key bits
bool isVulnerableBottomSubKey;	True if this instruction is marked as a value produced by the key scheduler, within the last ones to include all of the user key bits
bool isSubKey;	True if this instruction is represents an output of the key scheduler.
long PlaintextHeight;	If this instruction is in the core of the algorithm, so has met the plaintext, counts how many hops there are form this to the plaintext entrance in the cipher.
long CiphertextHeight;	If this instruction is in the core of the algorithm, so has met the plaintext, counts how many hops there are form this to the ciphertext exit at the end of cipher.
InstructionSource origin;	Tracks if this instruction is an instruction produced by previous passes or is an instruction that already does masked computations
bool isSbox;	Tracks if this value is the result of a substitution box

bool hasToBeProtected_pre;	Tracks if this instruction needs to be protected, as determined by the attack surface quantification algorithm for the part of the cipher near the plaintext
bool hasToBeProtected_post;	Tracks if this instruction needs to be protected, as determined by the attack surface quantification algorithm for the part of the cipher near the ciphertext
bool hasMetPlaintext;	Tracks if this instruction has the plaintext as its ancestor in the data flow graph, so it is not part of the key scheduler
vector<bitset<MAX_KEYBITS» keydep;	This is the matrix produced by SDFA with dependencies from the user key.
bitset<MAX_KEYBITS> keydep_own;	This bitset represents the user key dependencies eventually introduced by this instruction
vector<bitset<MAX_SUBBITS» pre;	This is the matrix produced by SDFA with dependencies from the topmost part of the subkeys
vector<bitset<MAX_KEYBITS» pre_keydep;	This matrix contains the same information as the matrix in the field above, but expressed as dependency from the user key instead of the user key
bitset<MAX_SUBBITS> pre_own;	This represents the dependencies from the topmost part of the subkeys directly introduced by this instruction
vector<bitset<MAX_SUBBITS» post;	Those three fields have the same meaning to the above three fields but with respect to the bottom part of the algorithm
vector<bitset<MAX_KEYBITS» post_keydep;	
bitset<MAX_SUBBITS> post_own;	
StatisticInfo keydep_stats;	Minimum, maximum and average number of dependencies from the three keys, calculated from the above matrices
StatisticInfo pre_stats;	
StatisticInfo post_stats;	
bitset<MAX_VALBITS> deadBits;	If this instruction represents a value returned by an S-BOX, keep track of eventual ”constant bits“.

Instruction* unmasked_value;	If this instruction is an already masked one, keep the pointer to the original unmasked one.
vector<Value*> MaskedValues;	If this instruction is an unmasked one and there are already the equivalent masked ones, keep here pointers to the shares representing this value.

Following the previous passes, the constant bit detection pass for SBOXes checks for bits that never change value for all entries in the substitution box, being it always 0 or always 1. This is useful to check the correctness of SBOXes and/or detect if their output are correctly reordered, especially for smaller 4-to-6 bit boxes like the one of DES. The algorithm, as described in the first part of this Chapter is then run on the IR. The SDFA algorithm was written using templates, enabling it to run with different *keys* for different parts of the attack surface quantification algorithm, reducing code-duplication. Most functions that operate on sets of instructions and discover nodes as they run are written as lambda function, passed to an executor function that runs them in parallel on independent sections of the graph using C++11 `future` mechanism, allowing it to take advantage of modern multi-core processors. The SDFA and masking algorithms were written using LLVM instruction visitor pattern, for added clarity and extensibility. It allows to define a default behavior for any type of instruction and override it only for some opcodes. The masking of tabulated SBOXes is accomplished by injecting a new function on the module containing the function being masked.

The algorithm for a 256-item SBOX masked with order 2 is:

```
void __sbox_masked(uint8_t original_sbox[],uint8_t share0,
     unit8_t share1,uint8_t share2, unit8_t *new_share0,
     unit8_t *new_share1, unit8_t *new_share2){
       uint8_t sbox_masked[256];
       *new_share1 = rand();
       *new_share2 = rand();
       for (i = 0; i<256; i++){
         sbox_masked[i^share1^share2]= \
         original_sbox[i]^(*new_share1)^(*new_share2);
       }
       *new_share0= sbox_masked[share0];
}
```

The original accesses to those boxes are written as a `GetElementPtr` and a `Load`. Both of them get replaced with:

- `alloca` for stack space for the new shares. (reused on subsequent calls)
- `call` to the `__sbox_masked` function.
- one `load` for each new share from the space allocated on the stack to an SSA register.

There is also a LLVM pass dedicated to producing output informations from informations contained in table 3.3. It has three different output formats: GraphViz, CSV and HTML. The Comma Separated Values file contains most of those informations ready for further automated elaboration and plot generation. GraphViz is a widely used format for graphs, and represents the data flow between instructions, classifying them with different colors. It is used to generate a SVG file with `dotty`. The HTML output generates a file for each single IR instruction, containing all the matrices expanded in an user-friendly way. For space reason, they are written by encoding in base64 the content of their memory, and JavaScript code will read it and plot in in a HTML5 Canvas. The entire HTML5 application allowing to browse the Data Flow Graph and examining details of each single instruction by clicking on it can be seen in figure 3.5

The source code is available on `http://github.com/maxximino/llvm` and `http://github.com/maxximino/clang` public repositories.

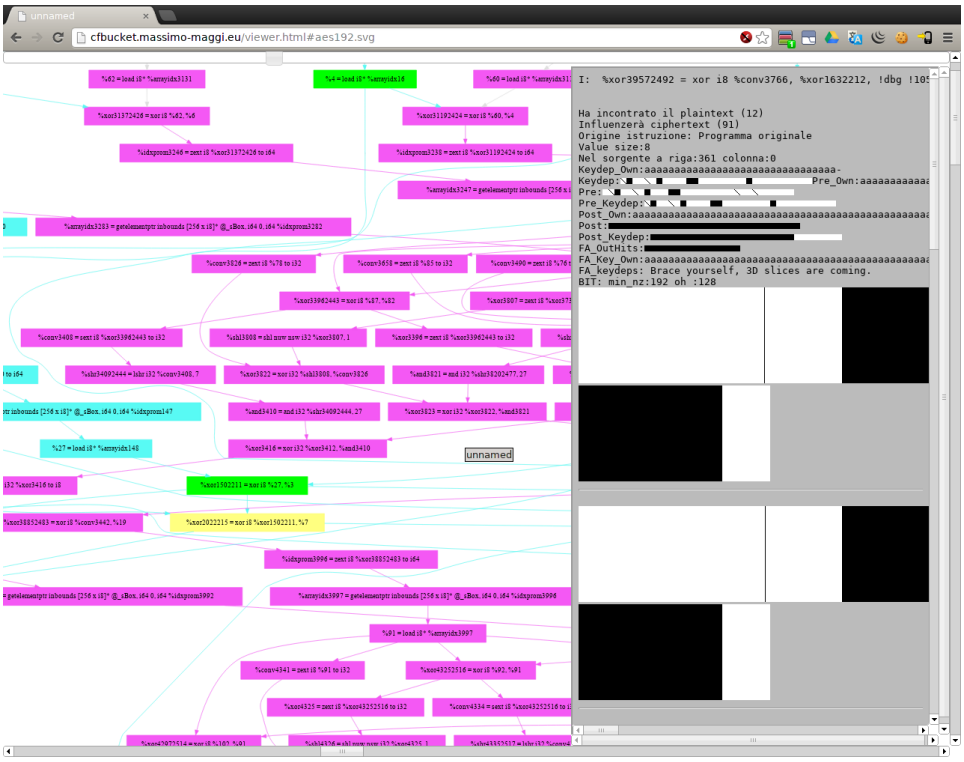


Figure 3.5: Screenshot of the web application created to browse results produced

Chapter 4

Experimental results

This chapter will contain data obtained from the automated procedure described in the previous chapter for the most common block ciphers.
There are plots of the instruction resistance for each block cipher considered, with data organized by cipher round. For the block ciphers without arithmetic operations there are also the performance results of the corresponding protected versions, examined on three different platforms.
These protected cipher implementations are split by the masking order and the security margin, which is the threshold for the instruction resistance value under which the masking countermeasure is applied.
A performance limit for the protected versions is given by the throughput of the random number generator which supplies the masks, so we'll examine how their throughput correlates to the cipher ones, highlighting novel insights on the practical feasibility of state-of-the-art countermeasures.

4.1 Results of Security Analysis

This section contains the detailed analysis of the vulnerability of instructions on each cipher, showing how fast it can diffuse key material up to the point of not being vulnerable to side channel attacks anymore. The instructions in the DFG of each cipher are depicted as a two dimensional colormap, their round number and the instruction position inside the round. Each instruction has a different color depending on its vulnerability.
The distinction in rounds is qualitative as the compiler optimizations may

move or merge instructions between different rounds.

4.1.1 AES

Figures 4.1, 4.2 and 4.3 report the results of our analysis applied to the 128-bit, 192-bit and 256-bit key variants of the AES.
The key is fully diffused only at the end of the third round (counting the initial extra *AddRoundKey* as round 0): this can be ascribed to the combined action of the diffusion layer of AES, which spreads the effect of a single bit change over a quarter of the state at each round, and the bytewise operating key schedule of the cipher. In particular, should be noted that, since the round key size of the AES is 128 bit, and the first round key employed is exactly the user key, both the 192- and the 256- bit variants of the AES take two full round to add the whole key material to the state and one extra to complete the diffusion. Note that the bytewise oriented operations of AES, which represented a winning design choice in terms of efficiency, turn out to be a factor causing a slow diffusion of the key material over the state, which is detrimental from the side channel attack point of view. Such an issue can be compensated through the use of a more aggressive key schedule, instead of a straightforward copy of the user key bits as the first round keys, in turn speeding up the diffusion of the whole key material over the state.

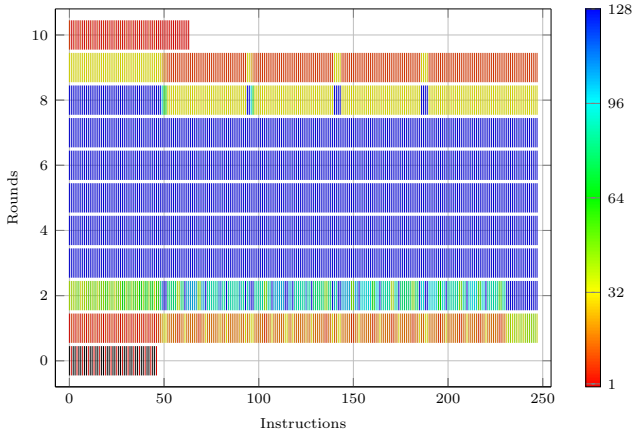


Figure 4.1: Per-instruction vulnerability of AES128

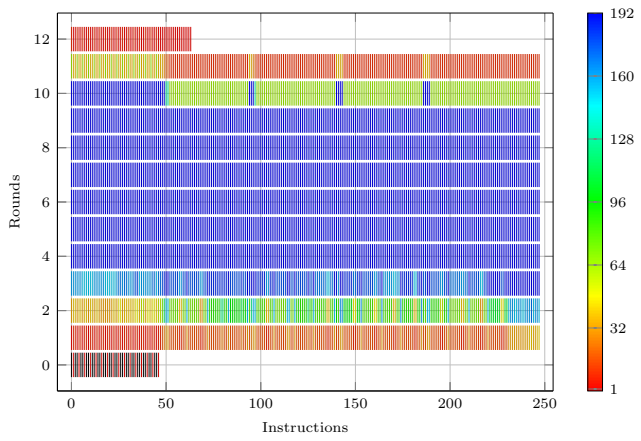


Figure 4.2: Per-instruction vulnerability of AES192

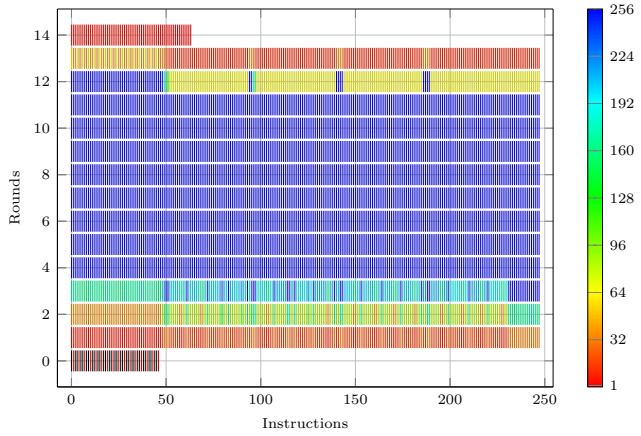


Figure 4.3: Per-instruction vulnerability of AES256

4.1.2 SERPENT-128

The strong asset of the Serpent cipher, from a side channel attack resistance point of view, is its key schedule as the computed the round keys are in such a form that that every bit of each round key depends on 64 bits of the user key. This property, paired with the efficient diffusion layer results in a quick convergence of the instruction resistance to the maximum value,

as shown by the analysis in Figure 4.4. Consequently, only a small subset of the Serpent cipher instructions need to be protected from side channel attacks. In a context where such protection is mandatory, it is likely that the Serpent cipher would outperform the AES once the overhead brought in by the necessary countermeasures is taken into account.

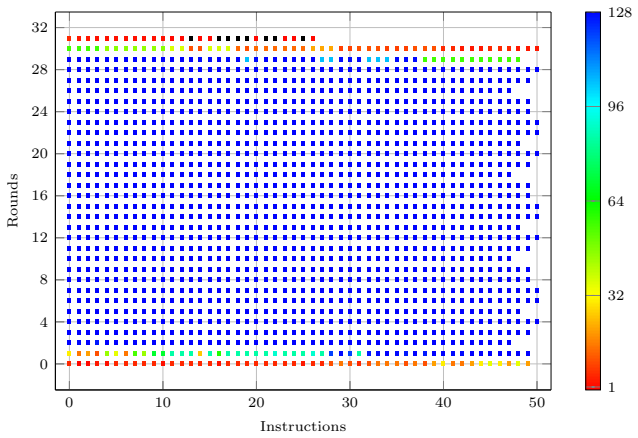


Figure 4.4: Per-instruction vulnerability of Serpent-128

4.1.3 Camellia

As shown in Figure 4.5, the Camellia algorithm has a good diffusion rate of key dependencies. However, the key schedule appears to be flawed, as in several points (the lighter vertical stripes on the right) instructions in all rounds are vulnerable to side channel attacks. This issue is due to a lack of key material mixing in some of the subkeys, among which, one is directly constituted of user key material without modifications (the instruction adding it is highlighted in red). This information is non-trivial, and to the best of our knowledge it is shown for the first time here, as the manual analysis of the algorithm did not allow its detection, due to the complex structure of the key schedule, which prevented easy inspection. This result underlines the relevance of designing and implementing automated tools for security analysis, as issues in the structure of a cipher may not be easily detected otherwise.

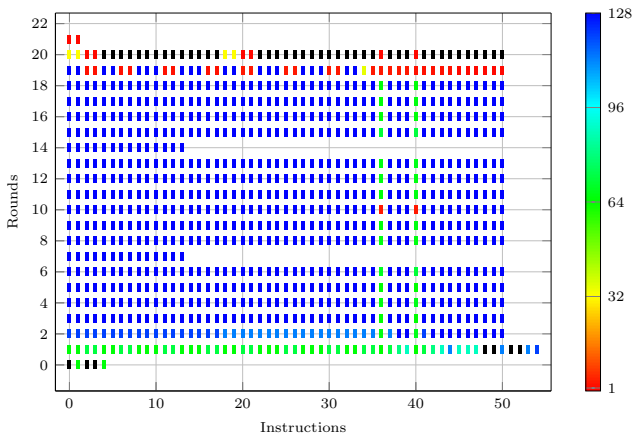


Figure 4.5: Per-instruction vulnerability of Camellia

4.1.4 DES and DES-X

Figures 4.6 and 4.7 report the vulnerability analysis of DES and DES-X. It can be seen that the key diffusion of DES is faster than the one of the AES: overall, 73.9% of DES instructions are protected by at least 40 bits of the key, whereas only 66.2% to 81.4% are protected in this way in AES (depending on the key size). DES-X has a higher diffusion rate than DES, as the extra key material with respect to the common DES is applied entirely in two 32-bit XOR operations, thus benefiting from the good diffusion provided by the DES.

Note that the fact that DES-X does not achieve an effective key length equal to the sum of the sizes of the three keys is detected by the analysis, which shows that the number of key bits protecting each intermediate value is never higher than 120. This result matches the common notion on DES-X, regarding the fact that its actual security margin is lower than 184 bits.

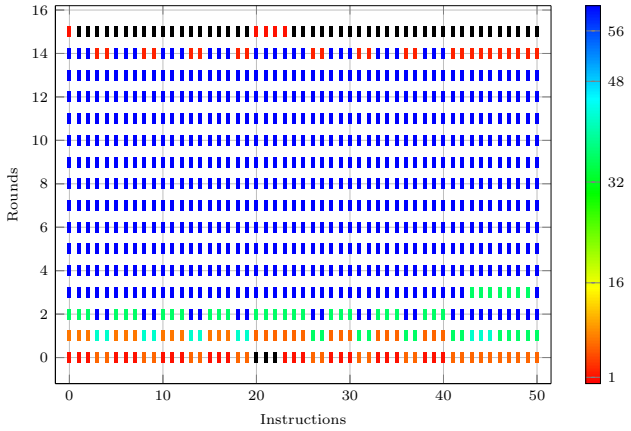


Figure 4.6: Per-instruction vulnerability of DES

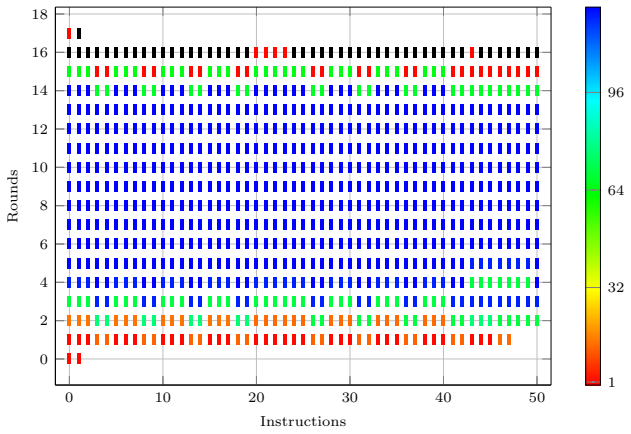


Figure 4.7: Per-instruction vulnerability of DES-X

4.1.5 3DES

Figures 4.10 and 4.11 reports the vulnerability analysis of Triple DES using keying options 2 (double key) and 1 (triple key), showing pairs of rounds rather than single rounds on the y axis for the sake of readability. It can be easily seen that Triple DES spectacularly fails at providing significant protection against side channel attacks, as it takes about two thirds

of the rounds to reach the full security margin. This is due to the fact that cascading multiple applications of the same cipher does impact adversely on the vulnerability to side channel attacks, as the security margin obtained is the same as the original cipher. For the same reason, there is no difference (besides the size of the key) between using Triple DES with keying option 1 or 2, from the point of view of side channel vulnerability, as the vulnerability of the first rounds is always the same as a single DES. In conclusion, it can be seen that, to harden Triple DES implementations against side channel attacks, it is necessary to apply countermeasures to about two thirds of the instructions in the cipher, which makes it rather demanding in terms of resources when compared to other ciphers.

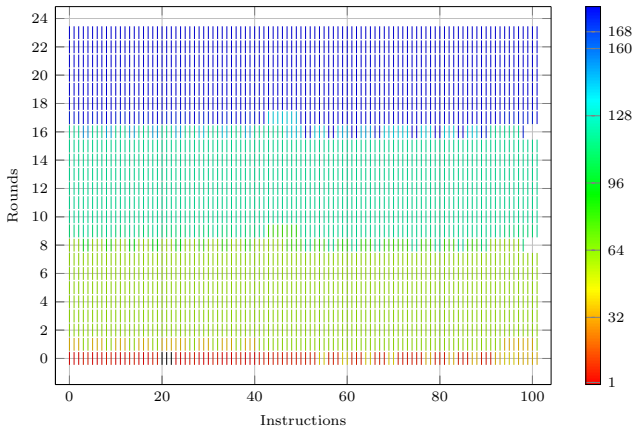


Figure 4.8: Per-instruction vulnerability analysis of 3DES3 with only forward analysis applied

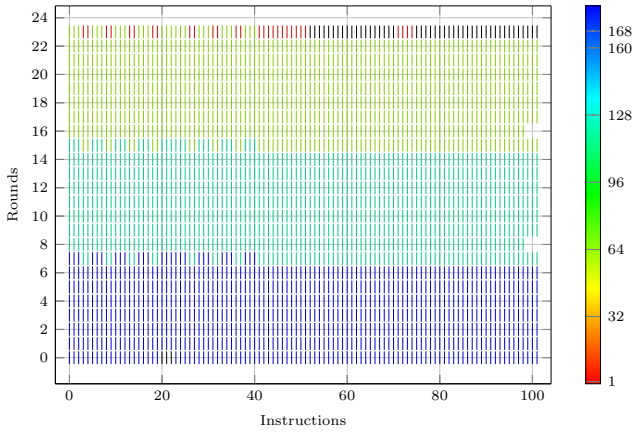


Figure 4.9: Per-instruction vulnerability analysis of 3DES3 with only backward analysis applied

The depiction of the forward analysis of TDES3 (keying option 1, figure 4.8) shows the difference w.r.t. keying option 2, for which the backward analysis 4.9 does not add any information. The ability to attack the cipher from the bottom (which is the reason for employing the backward analysis) prevents the additional independent key K_3 from having any beneficial effect from the point of view of side channel attack resistance.

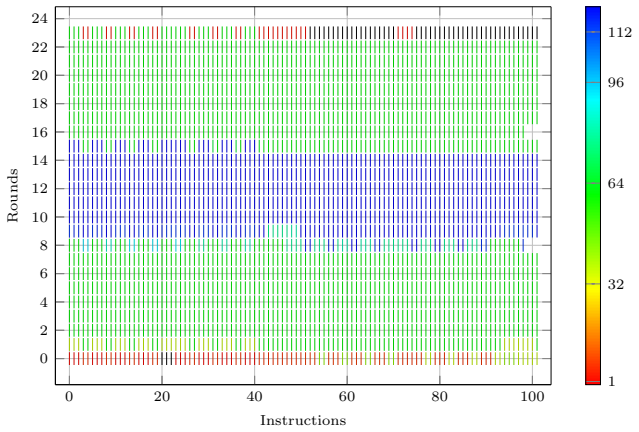


Figure 4.10: Per-instruction vulnerability of 3DES with keying option 2

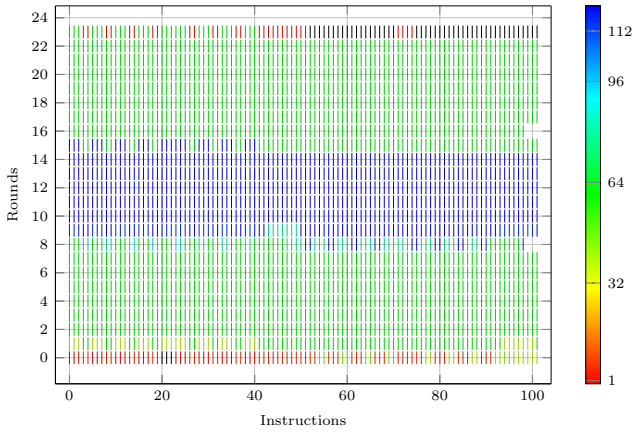


Figure 4.11: Per-instruction vulnerability of 3DES with keying option 1

4.1.6 GOST 28147-89

The simplicity of the cipher design (no permutations or expansions are present such as the ones in DES), combined with the absence of a key schedule (the user key is simply replicated to provide enough key material) makes it very vulnerable to side channel attacks. In particular, the lack of a key schedule significantly impacts on the key diffusion rate, which is provided only by the non-linear transforms of the Feistel function. The overall result is that eight rounds are needed to achieve full diffusion of the user key in the cipher state, which is coherent with the fact that eight rounds are needed to add 256 bits of key material in 32 bit chunks.

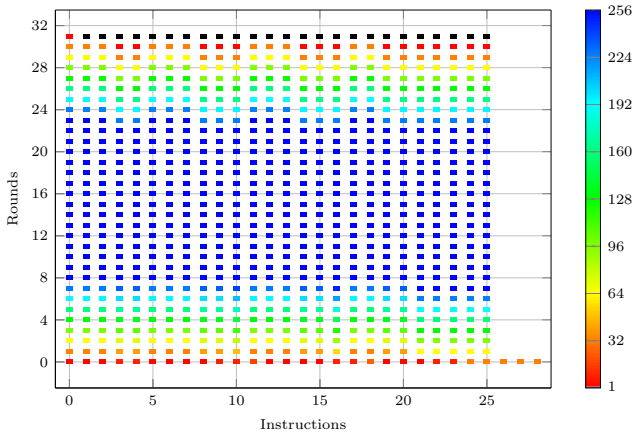


Figure 4.12: Per-instruction vulnerability of GOST 28147-89

4.1.7 CAST5

The use of arithmetic operations on which CAST5 relies is a key element for a fast diffusion of the key bits, and the strong key schedule obtains each round key bit from contributions from at least 56 bits of the original cipher key. These two factors make CAST5 one of the best ciphers examined from the key diffusion standpoint. Further cause of fast diffusion is the use of substitution maps that take 8 bits as input but produce 32 bits as output, thus diffusing the key influence on each input bit onto the 32 bits of output. The overall result is that the CAST5 algorithm is more resistant than any of the previous ciphers (except Serpent) to side channel attacks, and will therefore benefit from a reduced overhead when applying countermeasures, provided that an efficient way to protect arithmetic instructions is used.

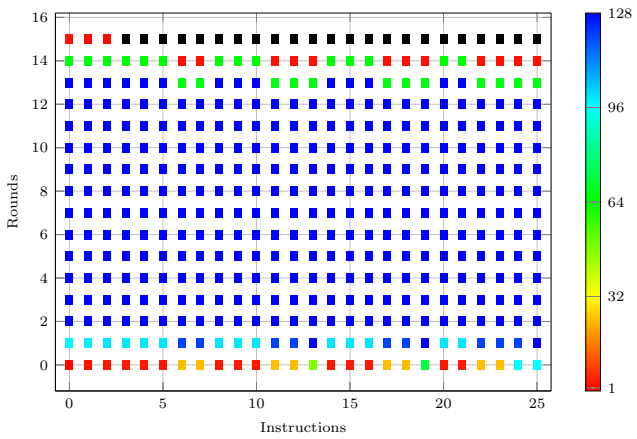


Figure 4.13: Per-instruction vulnerability of CAST5

4.1.8 Conclusion

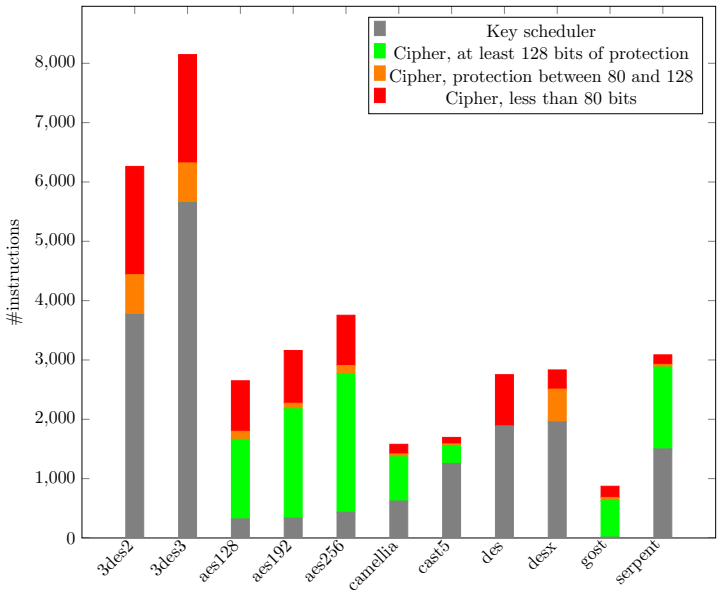


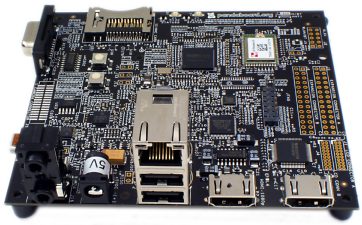
Figure 4.14: Overview of the number of instructions needing protection in each cipher analyzed

Figure 4.14, compares the relative code size of the considered block ciphers and allows to both evaluate their vulnerability and assess the expansion of the code size when applying the countermeasures.
The plot provides also the detail on the entity of the code which actually needs protection. In particular, the gray part of the bar is the key scheduler, which is not vulnerable to DPA attacks [17] as it only elaborates secret values.
The part of the cipher doing the real encryption, i.e. dealing with the plaintext, is colored differently depending on the vulnerability level. The green is used for 128+ bit-protection, orange for protection between 127 and 80 bits and red for less than 80 bit protection. It is interesting to note that, a loop-unrolled version of the 3DES is twice as big as the AES256 and does not provide a comparable security margin. This diagram shows that most of the code size 3DES is consumed by the key scheduler. Viceversa, this chart shows that the cipher with the lowest percentage of code that needs protection is Serpent, as seen as in Section 4.1.2. The AES cipher, due to its fast diffusion and small key scheduler, has the highest percentage of already 128-bit-safe code. The total absence of a key scheduler in the GOST algorithm has a favourable impact on the code size, albeit at the cost of a slow diffusion, but the smallest cipher without considering the key scheduler (as it never gets masked) is CAST5.

4.2 Experimental Workbench

There are three test setups for three different architectures:

The *x86_64* tests were executed on a desktop computer with a AMD Phenom(tm) II X6 1090T processor running at 3.4GHz, with 128 KB L1 cache, 512 KB L2 cache, and 6MB L3 cache shared between cores. There are 16GB of DDR3 RAM. The storage section is composed by 4 WD Caviar Black used as two couples of ZFS mirror vdevs. The operating system is Gentoo Linux with kernel 3.9.4 and GLIBC 2.17 . Every binary was compiled with options `-O3 -march=x86-64` (detailed compilation procedure in listing A.3)



The *arm.a9* tests were executed on the *Pandaboard ES development platform*. The SoC is a Texas Instruments OMAP4460 SoC[1], containing a dual core Cortex A9-MPCore clocked at 1.2 GHz, 1 MiB L2 cache and 1 GiB DDR2 RAM, running Linaro 12.09 Linux distribution (`armv7l` target), with kernel 3.4.0 and EGLIBC 2.15. The system runs on a 4GB SDHC class 6 . Every binary was compiled with options `-O3 -march=arm -mcpu=cortex-a9` (detailed compilation procedure in listing A.3)

The *arm.pogo* tests were executed on the *PogoPlug v2 pink*. The SoC is a Marvell 88F6281[2], with one core running at 1.2GHz, 256kB L2 cache and 256MB of DDR2 RAM, running Arch Linux with kernel 3.1.10 and GLIBC 2.17. The system runs on a Corsair 32GB pen drive. [Every binary was compiled with options `-O3 -march=arm -mcpu=arm926ej-s` (detailed compilation procedure in listing A.3)

4.3 Passive SCA Protection

Masking, as a way to protect from passive SCA attacks, is an already established technique whose limits where related mainly to huge performance loss and code size increase. This section describes how this new analysis method allowed us to improve performance and reduce the increment in code size of masked algorithms.
The algorithms GOST and CAST5 were not tested, as they contain arithmetic operations. It is possible to perform conversions between boolean

[1]Further information about the SoC can be found at `http://www.ti.com/litv/pdf/swps046a`

[2]Further information about the SoC can be found at `http://www.marvell.com/embedded-processors/kirkwood/assets/88F6281-004_ver1.pdf`

masked values and arithmetic masked ones [10], to apply correct and performant arithmetic masking. This option is not incompatible with our implementation but has not been implemented.

4.3.1 Performances

The throughput is calculated by repeatedly measuring the execution time of 1.000.000 encryptions of a plaintext block, through employing the test code shown in listing A.1 (page 103) compiled with instructions at listing A.3 on page 104.
The throughput is defined as

$$\frac{\text{BLOCK SIZE} \cdot \text{ITERATIONS}}{\frac{\sum \text{EXECUTION TIME}}{\text{NUMBER OF SAMPLES}}}$$

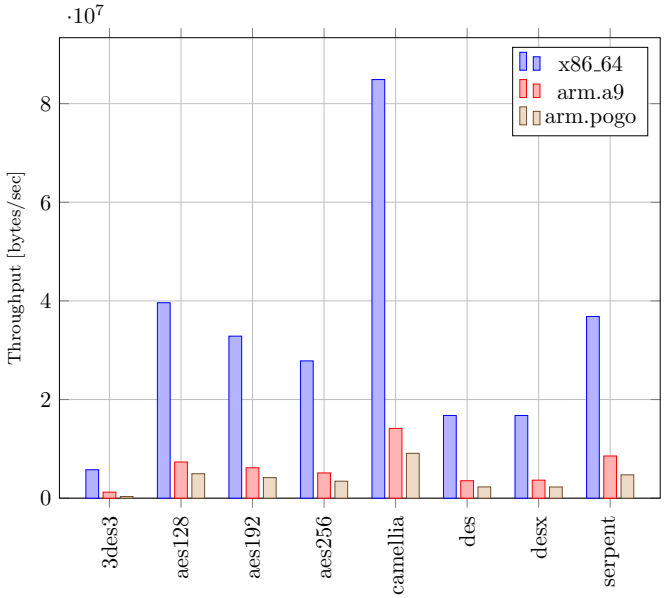


Figure 4.15: Throughput of reference implementations on reference platforms

Figure 4.15 shows the throughput obtained on our reference platforms by the unprotected versions of the algorithms compared. The AES cipher implementation chosen, from this chart, does not look optimal as it is even

slower than Serpent.Usually, fast AES implementations use T-Tables, which are tables of precomputed values to be reused through the computation. We chose a direct implementation of the specifications because it allowed us to fine-tune SDFA with a wider range of operations. The implementation chosen is optimized for 8-bit microcontrollers, because AES operations are specified at byte level. It's possible to notice how the different algorithms relative throughput stays unchanged among test platforms. The highest performing one is Camellia, followed by Serpent and the AES family. This fact is not unexpected, as DES was designed to be extremely efficient when implemented in hardware then it's not so fast in software implementations. Its derivated algorithms obviously inherit this property. As previously said, our AES implementation is not optimized for performance.

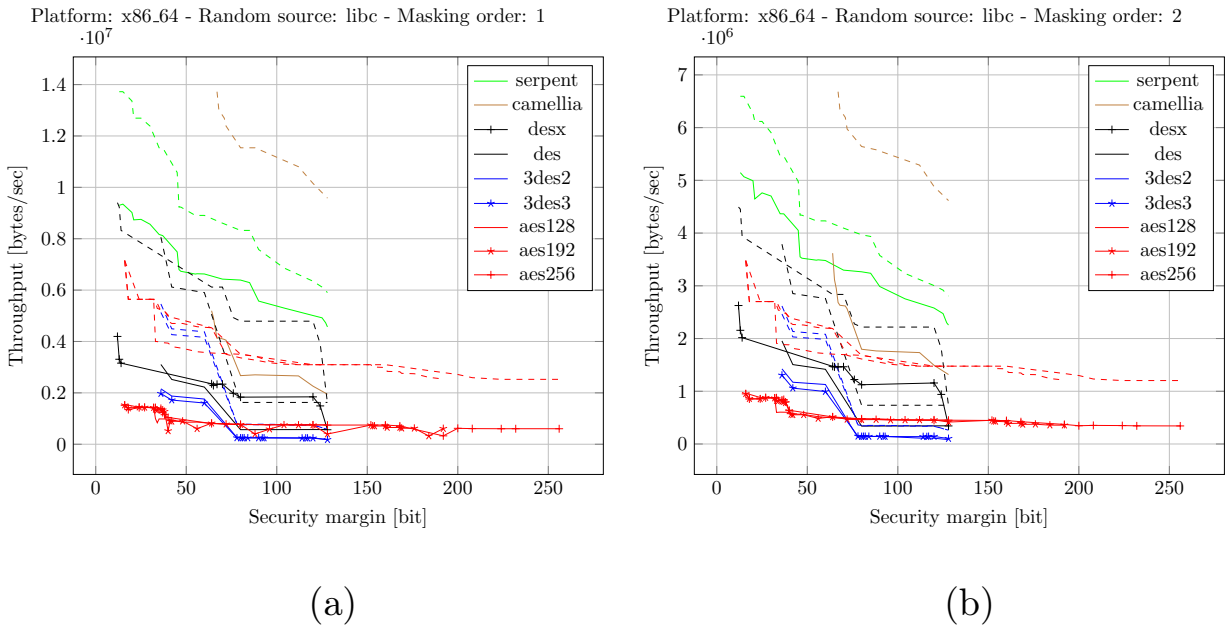
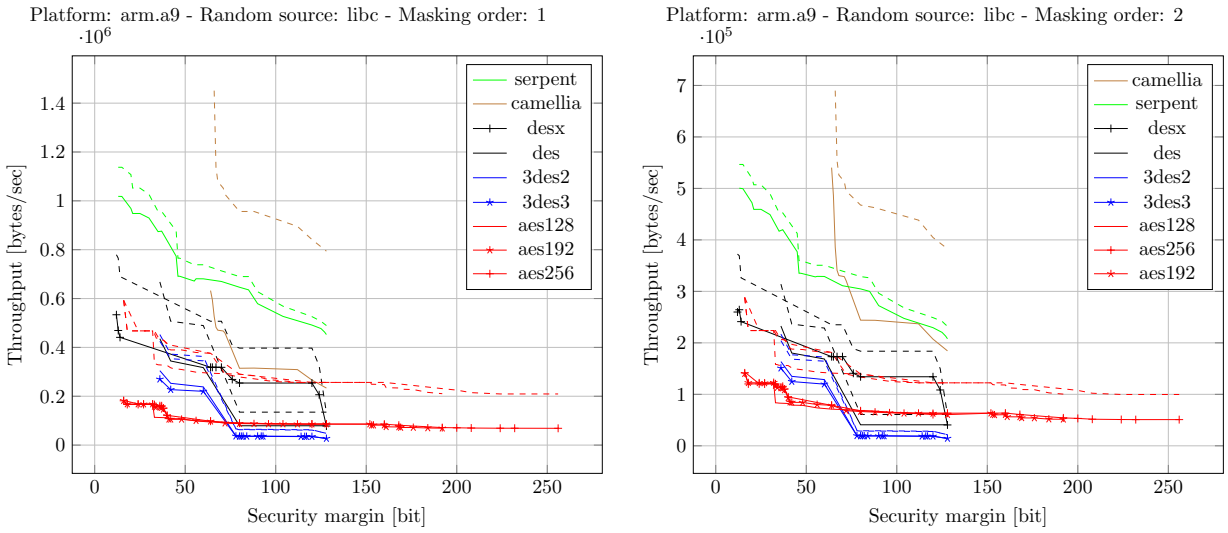
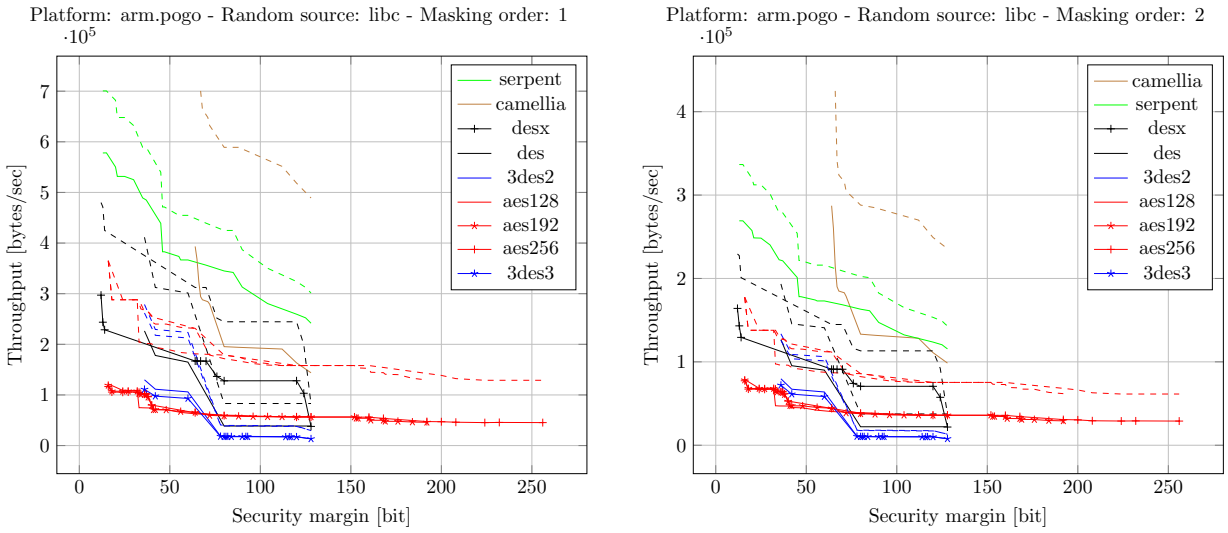


Figure 4.16: Throughput on *x86_64* with `rand()` overhead



(a) (b)

Figure 4.17: Throughput on *arm.a9* with `rand()` overhead

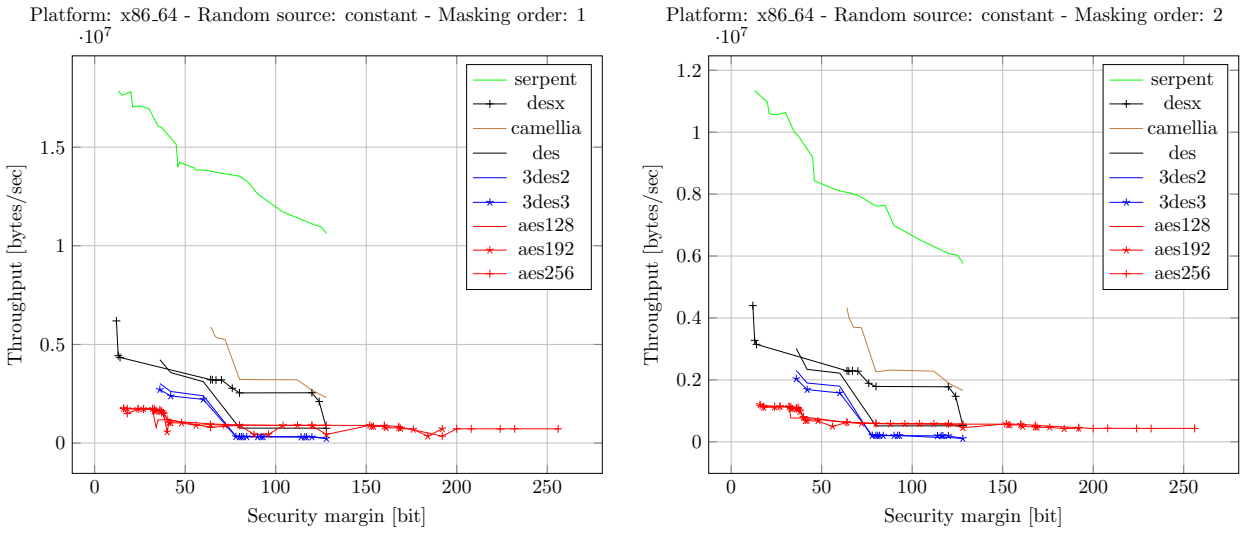


(a) (b)

Figure 4.18: Throughput on *arm.pogo* with `rand()` overhead

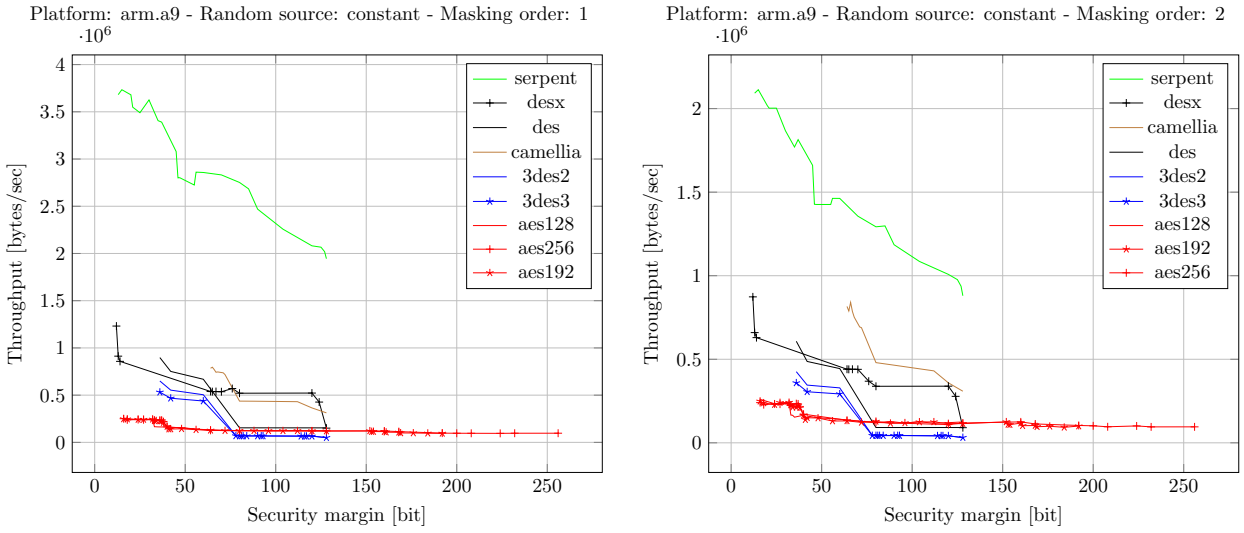
Figures 4.16a, 4.16b, 4.17a, 4.17b, 4.18a and 4.18b depict the performance data of all possible combinations of algorithms, platforms and masking

order.
Moreover, the dotted lines represent the corresponding maximum theoretical throughput limit, computed from data in section 4.3.2. This maximum theoretical throughput limit is given by libc's `rand()` implementation, and could be overridden by using a different and more performing random number generator.
It's important to note how the throughput of most algorithms drops in clearly visible steps. Those steps correspond to the crossing of a *diffusion* operation, which is expensive (in terms of performance) to mask, and after that the rest of the round has about the same security level of that operation. So it's sensible that the algorithm chooses to mask almost an entire round.
The AES performance is hindered by the non-optimal implementation as the basic S-BOX version was employed. However the three variants of AES have really similar throughput.
The most bounded algorithm by the RNG performance, as shown in previous section, is Serpent, as its throughput is the closest to its maximum throughput limit. It is interesting to notice in comparing ARM-based platforms to *x86_64*, one should notice how Camellia performs better compared to others. This behavior can be explained by the high number of bit-shift operations in Camellia algorithm and the fact that those operations in ARM architectures are executed by the barrel-shifter, in the same pipeline as the next ALU operations, saving a significant amount of processing cycles. [4]



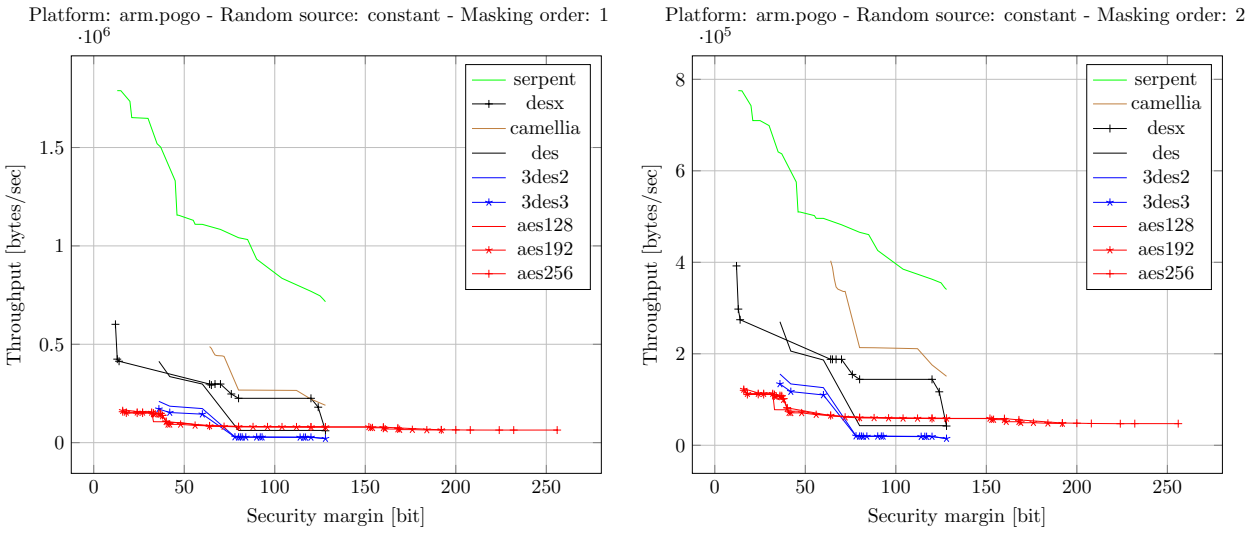
(a) (b)

Figure 4.19: Throughput on *x86_64* without `rand()` overhead



(a) (b)

Figure 4.20: Throughput on *arm.a9* without `rand()` overhead



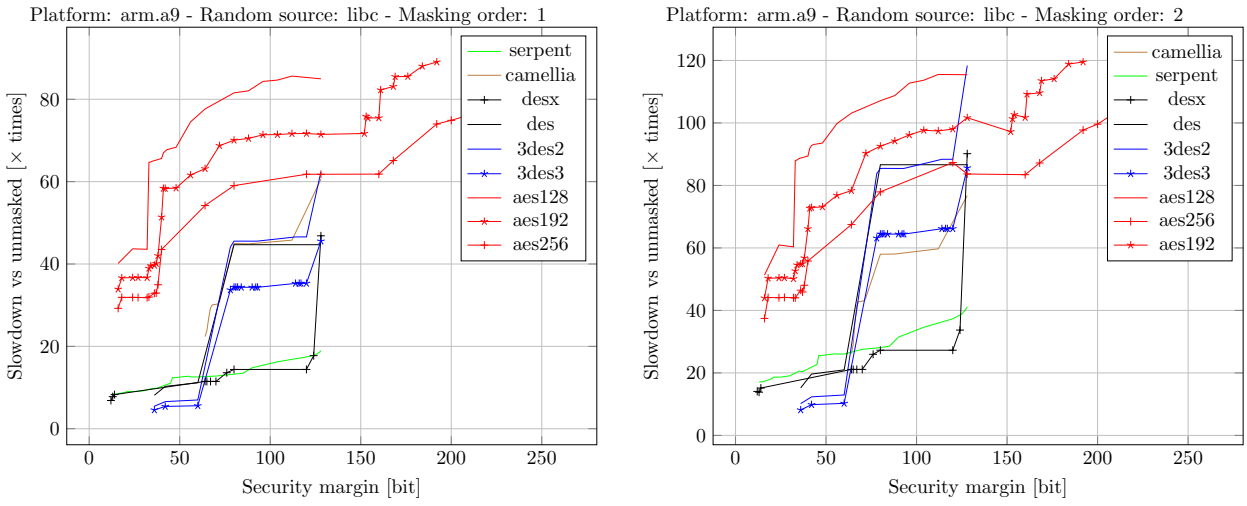
(a) (b)

Figure 4.21: Throughput on *arm.pogo* without `rand()` overhead

When looking at results without `rand()` overhead, obtained replacing `rand()` calls with a constant, the most evident effect is that Serpent throughput is clearly improved. This is another confirmation of the fact that its masking needs a rather constraining amount of random values. The reason behind this fact is that Serpent does not use SBOX for diffusion, but uses a set of non-linear bitwise functions. Masking for lookup tables uses few randoms repeatedly (and is not secure for use with with masking order greater than 2) for a lot of computations, while a bitwise function requires some new random values for each instruction executed.
Slowdown is defined as

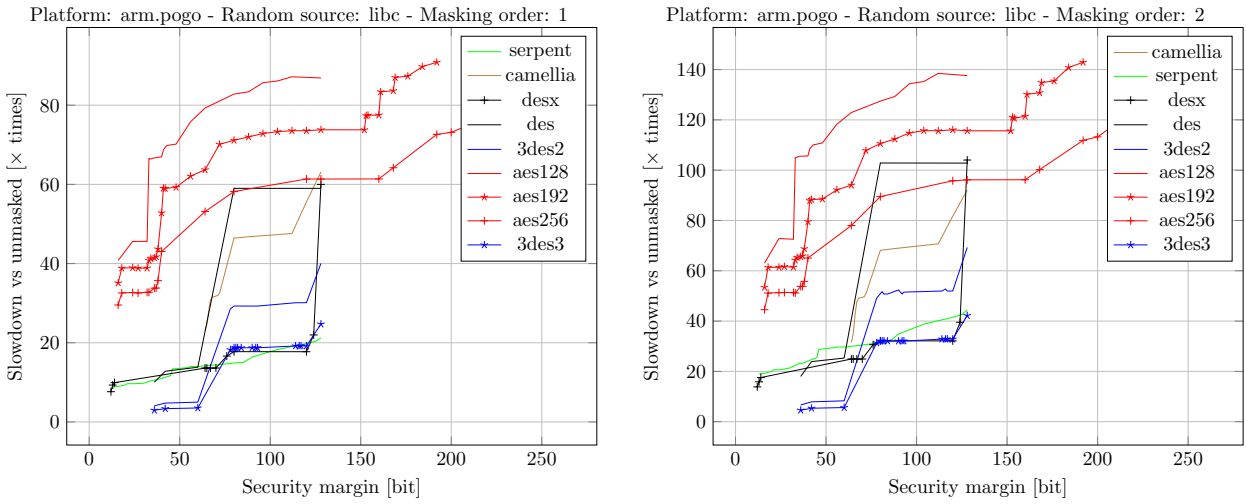
$$\frac{\text{Throughput of unmasked algorithm}}{\text{Throughput of masked algorithm}}$$

.



(a) (b)

Figure 4.22: Slowdown on *arm.a9* with `rand()` overhead



(a) (b)

Figure 4.23: Slowdown on *arm.pogo* with `rand()` overhead

Figures 4.22a, 4.22b, 4.23a and 4.23b depict the data regarding the slowdown relative to the unmasked cipher. AES has great slowdown as it is

an 8-bit optimized implementation, so it's composed by a large number of instructions.
In the 3DES family it is possible to notice at which security margin entire DES algorithms get masked, as there is a steep increase in the overhead. The algorithm least impacted by masking is Serpent, which, however, is the most sensitive to an insufficient RNG throughput. The results show that its throughput is going to scale well with respect to RNG throughput. When comparing relative slowdowns, there are no significant differences between the two ARM platforms.

4.3.2 Random Number Generation Issues

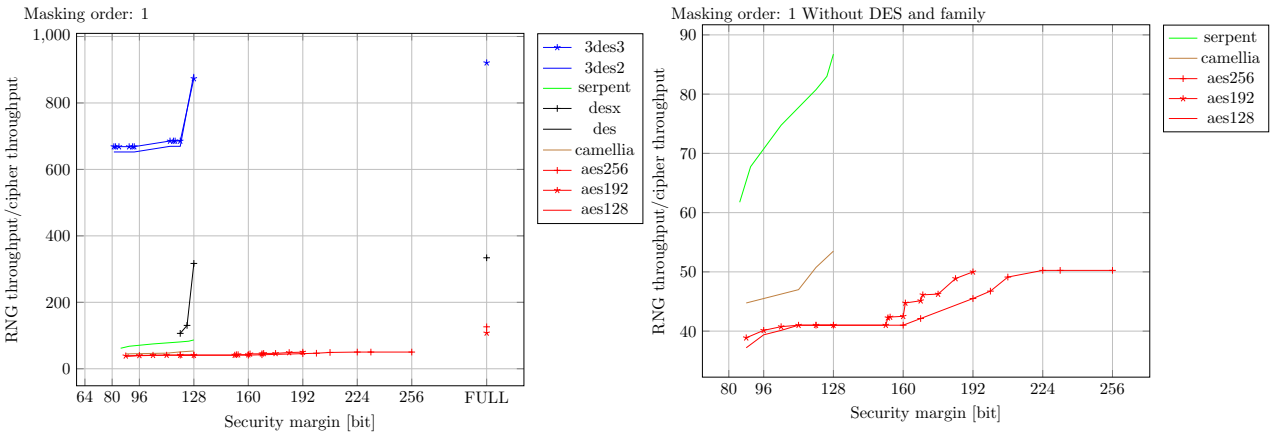
This quantity of entropy needed for each algorithm was determined by counting the calls to the `rand()` function during the encryption of a single block.
The entropy need at a masking order and chosen security level is defined as

$$\frac{4(\text{BYTES}) \cdot \text{NUMBER OF CALLS TO } \texttt{RAND()}}{\text{ALGORITHM BLOCK SIZE}}$$

.
In practical terms, this is equal to

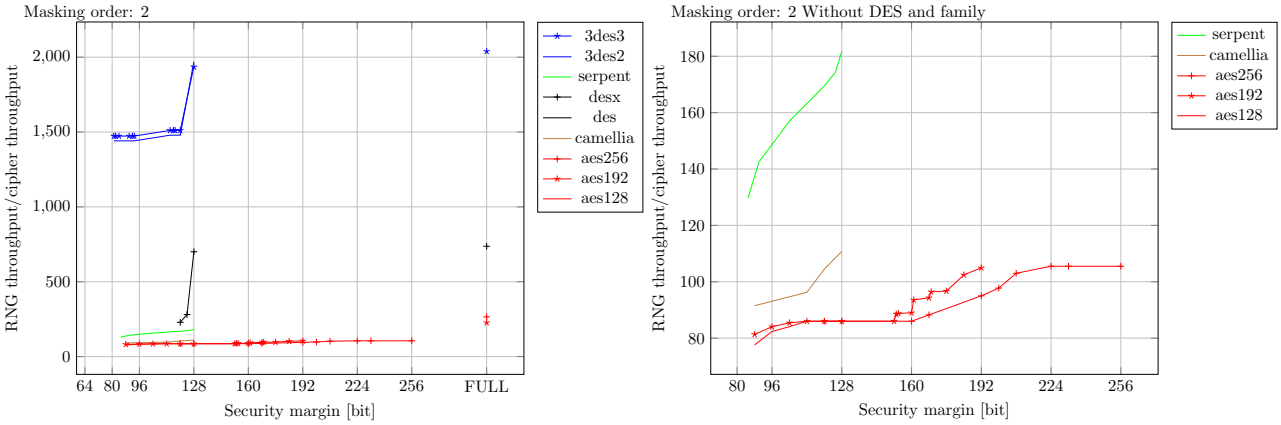
$$\frac{\text{RNG THROUGHPUT}}{\text{CIPHER THROUGHPUT}}$$

.
The count of the calls to the `rand()` function was obtained by using the *LD_PRELOAD* mechanism with a library with the code in listing A.2. The GNU/Linux dynamic linker can load a library (specified in *LD_PRELOAD* environment variable) before others, allowing to override symbols from other libraries.An example of the practical setup to do so is available at page 104, listing A.2.
A corrective coefficient was applied to AES family, as the analyzed implementation computes 8-bit values, thus it effectively discards 24 bits provided by each call to `rand()`.



(a) (b)

Figure 4.24: Entropy needs at masking order 1



(a) (b)

Figure 4.25: Entropy needs at masking order 2

The ciphers that need the highest quantity of random data are the DES-like ones, as they contain a huge number of instructions in need of masking when compared to other cipher designed with software implementations efficiency in mind. Their entropy needs are so large compared to other ciphers, that Figures 4.24a and 4.25a were replicated in Figures 4.24b and 4.24b without including them for the sake of clarity. Within the set of ciphers with

a reasonable need for entropy to be masked, Serpent is the least efficient as in our examined implementation it does not have tabulated S-BOXes: they are computed algebraically. This requires changing the random shares at each of the instructions, while masking a tabulated S-BOX uses the same random for a large number of different computations, as explained in 15.

Benchmark of libc `rand()` function

Table 4.1: `rand()` benchmark

Platform	Throughput
x86_64	484.22 MB/s
arm.a9	40.14 MB/s
arm.pogo	24.72 MB/s

From data in this table, it's evident that desktop platforms have an order of magnitude of advantage in random number generation through `libc` with respect to actual embedded systems. The higher frequency of the PogoPlug with respect to the PandaBoard, is not enough to compensate for its older *armv5* architecture instead of the newer *armv7l*. By comparison, Intel's *RdRand* instruction included in recent *Ivy Bridge* processors can produce more than 500MB/s of random data. An hobbyst-project employing a cheap FPGA (`http://hackaday.com/2010/02/06/hardware-based-randomness-for-linux/`) can produce up to 1MB/s of true random data, while the RNG embedded in ST ARM-based STM32F4xx microcontrollers can produce more than 4MB/s of random data.

Improvements in throughput versus traditional full masking

Table 4.2 shows throughput of fully masked ciphers and in comparison the optimized version with equivalent security margin, equal to the cipher key size. The data is presented only for the *arm.a9* platform, as the data for other platforms show a similar behavior. Throughput values (columns Full [masking] and Optimized [masking] are expressed in kiB/s), and the Relative column is

$$\frac{\text{Full masking throughput}}{\text{Optimized throughput}}$$

expressed in percent.

Table 4.2: Throughput improvements vs. traditional full masking on *arm.a9* platform

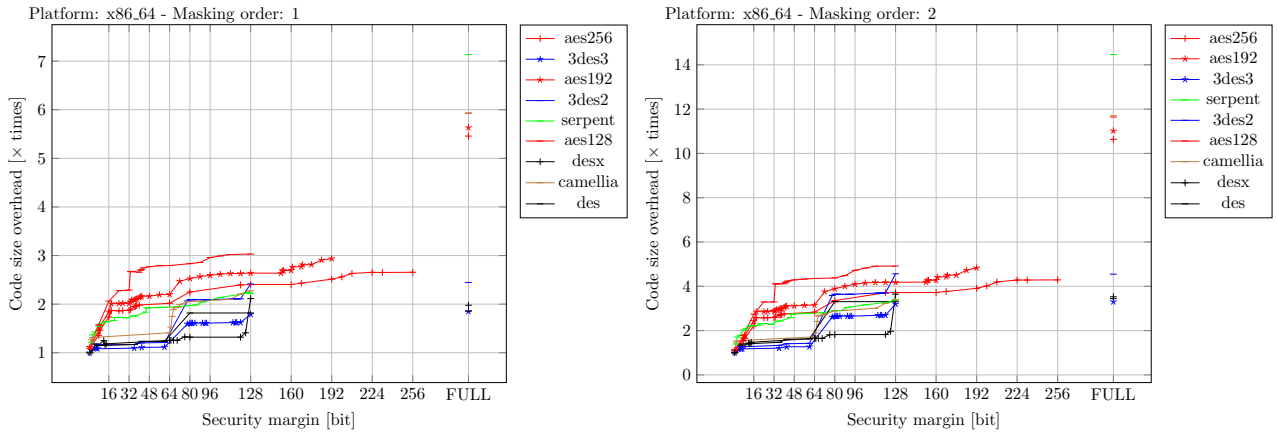
Algorithm	Masking order	Full	Optimized	Relative
3des2	1	25.7	34.9	135.6%
	2	13.6	18.4	134.7%
aes128	1	34.8	84.5	242.4%
	2	26.8	62.2	232.0%
aes192	1	28.9	67.6	233.7%
	2	22.1	50.4	228.1%
aes256	1	24.7	67.1	271.7%
	2	18.9	49.7	262.8%
camellia	1	44.6	228.0	511.0%
	2	36.7	180.1	490.9%
des	1	74.5	309.8	415.7%
	2	38.6	164.7	426.4%
desx	1	73.7	248.0	336.6%
	2	38.1	131.0	343.4%
serpent	1	69.1	442.0	639.5%
	2	32.2	203.1	630.0%

4.3.3 Code Size

Code size is particularly important in embedded environments, because today's CPUs should fit in increasingly smaller packages (think about smart cards, or microSIM) and their price should drop constantly.
Thus there are two pushes to reduce the area of the die, and resizing the memory is the most straightforward approach to lower die area.
Another reducible part of the die is the CPU cache, which is really important for performance. A smaller algorithm can fit in a smaller, cheaper cache.
Masking is going to expand code size, as we are not simply computing a given instruction: for each one we need to store (and execute) a group of instructions which work on multiple values for each original one and can be further elaborated or reduced to the originally requested value.
Code size was measured as the size of the *.text* ELF section of the test executable compiled with instructions at listing A.3 on page 104 The test executable code,other than the reference algorithm, is shown in listing A.1 on page 103
Overhead is defined as

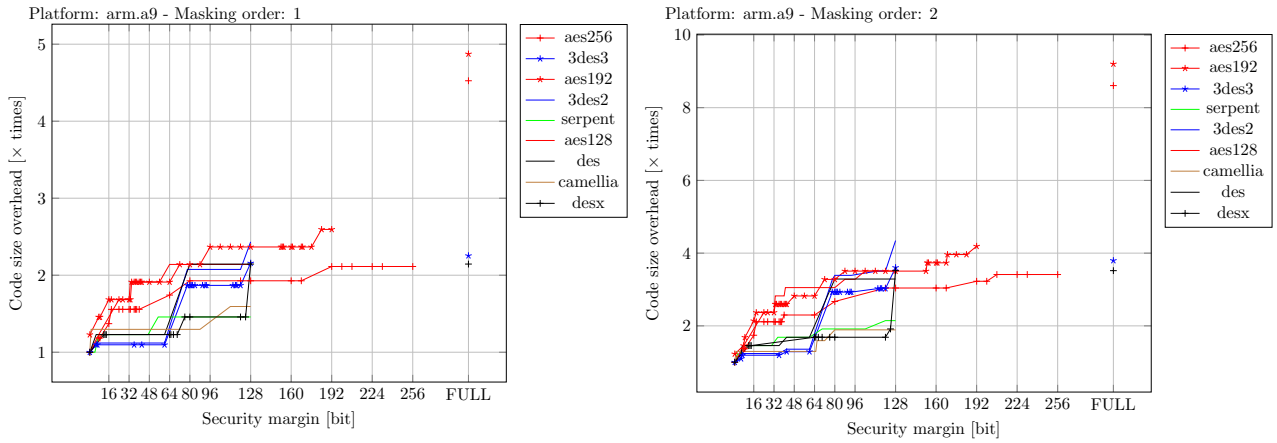
$$\frac{\textit{.text} \text{ ELF section size of masked algorithm}}{\textit{.text} \text{ ELF section size of unmasked algorithm}}$$

.



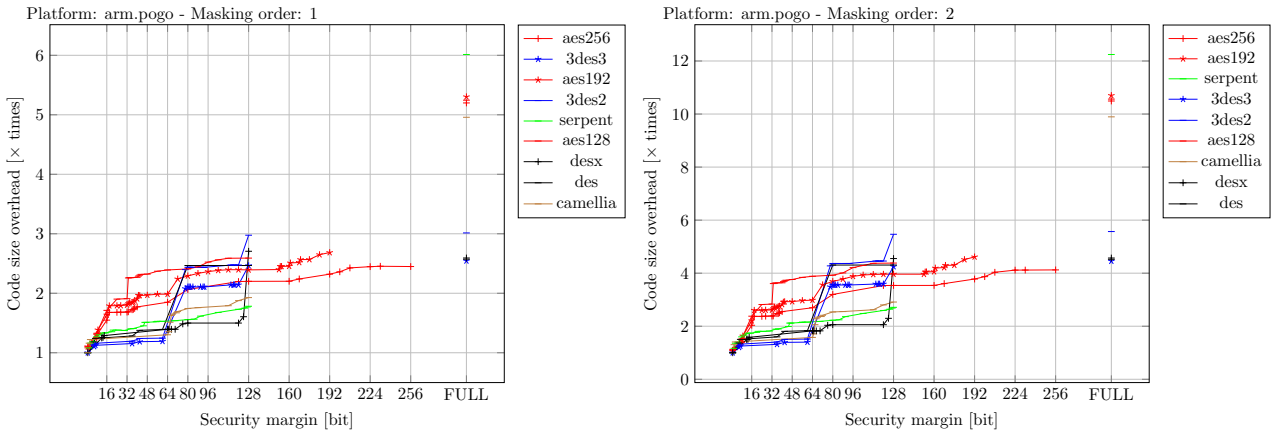
(a) (b)

Figure 4.26: Code size overhead on *x86_64*



(a) (b)

Figure 4.27: Code size overhead on *arm.a9*



(a) (b)

Figure 4.28: Code size overhead on *arm.pogo*

In Figures 4.26a, 4.26b, 4.27a, 4.27b, 4.28a and 4.28b, it's possible to notice how the proposed approach definitely limits the code size increase when compared to the usual full algorithm masking.
The most favorable is Serpent: with full masking at masking order 2, its size increases by a factor 12, while with the proposed algorithm the size increases by a factor less than 3
When selectively masked at 128-bit security margin (the same as its key length considered) its size increases by less than a factor 3, yielding a net 3× improvement.
AES has a slightly lower gain, and DES-family has the weakest improvement.

Chapter 5

Conclusions

This work has proposed an automated technique to perform in-depth analysis of the characteristics of block ciphers, and provide efficient and effective countermeasures against side channel attacks. The complete automation of the procedure provides an effective way to eliminate the risk of human errors in the otherwise complex and tedious process.
This is a cross-disciplinary work where analysis principles typically employed to optimize programs in compilers, were applied to produce a new analysis technique effective in another domain: applied cryptography.
This work has introduced the Security-Oriented Dataflow Analysis [1], which has been subsequently used in the Attack Surface Quantification Algorithm to analyze block ciphers. The proposed analysis and countermeasure application technique has proven its practical effectiveness through being employed to screen the instruction level security of six real world ciphers, as well as apply automatically strong side channel countermeasures with a fine grain. We obtained notable improvements in performance and code size of masked ciphers with an designer-tunable loss in security margin, which can be easily chosen at compile time with no effort from the designer. The finer grain of countermeasure application allowed by our technique achieves an over 6× improvement in throughput and a 4.5× reduction in code size, without any loss on security margin compared to current state of the art cipher protection techniques.

Appendix A

Appendix

Algorithm A.0.1: SERPENT-128 Encryption

Input: p, plaintext; k, cipher key
Data: $SBOX_{1..8}$, 8 substitution boxes
Output: c, ciphertext

begin
  $s \leftarrow p$
  $\langle SK^{(0)}, SK^{(1)}, \ldots, SK^{(32)} \rangle \leftarrow \textsc{KeySchedule}(k)$
  for $i \leftarrow 0$ **to** 30 **do**
    $s \leftarrow \text{SBOX}_{i\%8}(s \oplus SK^{(i)})$
    $s_1 \leftarrow \textsc{RotateLeft}(s_1 \oplus \textsc{RotateLeft}(s_0, 13) \oplus \textsc{RotateLeft}(s_2, 3), 1)$
    $s_3 \leftarrow \textsc{RotateLeft}(s_3 \oplus \textsc{RotateLeft}(s_2, 3) \oplus (\textsc{RotateLeft}(s_0, 13) \ll 3), 7)$
    $s_0 \leftarrow \textsc{RotateLeft}(\textsc{RotateLeft}(s_0, 13) \oplus s_1 \oplus s_3, 5)$
    $s_2 \leftarrow \textsc{RotateLeft}(\textsc{RotateLeft}(s_2, 3) \oplus s_3 \oplus (s_1 \ll 7), 22)$
  $s \leftarrow \text{SBOX}_7(s \oplus SK^{(31)})$
  $c \leftarrow s \oplus SK^{(32)}$
  return c

Algorithm A.0.2: CAST-5 Encryption

Input: p, plaintext block as 2-32 bit elements array.; k, cipher key
Data: $s_{1..4}$, 4 substitution boxes
Output: c, ciphertext block as 2-32 bit elements array

begin
  $l \leftarrow p^{(0)}$
  $r \leftarrow p^{(1)}$
  $\langle Km^{(0)}, Km^{(1)}, \ldots, Km^{(15)}, Kr^{(0)}, Kr^{(1)}, \ldots, Kr^{(15)} \rangle \leftarrow$ KeySchedule(k)
  for $i \leftarrow 0$ **to** 15 **do**
    $t = l$
    $l = r$
    switch i%3 **do**
      case *0*
        $\langle I^{(0)}, I^{(1)}, I^{(2)}, I^{(3)} \rangle \leftarrow$ RotateLeft$(Km^{(i)} + r, Kr^{(i)})$
        $r = t \oplus (s_1(I^{(0)}) \oplus s_2(I^{(1)}) - s_3(I^{(2)}) + s_4(I^{(3)}))$
      case *1*
        $\langle I^{(0)}, I^{(1)}, I^{(2)}, I^{(3)} \rangle \leftarrow$ RotateLeft$(Km^{(i)} \oplus r, Kr^{(i)})$
        $r = t \oplus (s_1(I^{(0)}) - s_2(I^{(1)}) + s_3(I^{(2)}) \oplus s_4(I^{(3)}))$
      case *2*
        $\langle I^{(0)}, I^{(1)}, I^{(2)}, I^{(3)} \rangle \leftarrow$ RotateLeft$(Km^{(i)} - r, Kr^{(i)})$
        $r = t \oplus (s_1(I^{(0)}) + s_2(I^{(1)}) \oplus s_3(I^{(2)}) - s_4(I^{(3)}))$
  $c^{(0)} \leftarrow r$
  $c^{(1)} \leftarrow l$
  return c

Algorithm A.0.3: GOST 28147-89 Encryption

Input: p, plaintext block as 2-32 bit elements array.; k, cipher key as 8 32-bit elements array; $SBOX$, chosen substitution boxes
Output: c, ciphertext block as 2-32 bit elements array

begin
  $r \leftarrow p^{(0)}$
  $l \leftarrow p^{(1)}$
  for $i \leftarrow 0$ **to** 2 **do**
    for $j \leftarrow 0$ **to** 3 **do**
      $r \leftarrow r \oplus$ RotateLeft(SBOX$(l + k^{(2j)})$,11)
      $l \leftarrow l \oplus$ RotateLeft(SBOX$(r + k^{(2j+1)})$,11)
  for $j \leftarrow 3$ **to** 0 **do**
    $r \leftarrow r \oplus$ RotateLeft(SBOX$(l + k^{(2j+1)})$,11)
    $l \leftarrow l \oplus$ RotateLeft(SBOX$(r + k^{(2j)})$,11)
  $c^{(0)} \leftarrow l$
  $c^{(1)} \leftarrow r$
  return c

Algorithm A.0.4: Camellia Encryption

Input: $\langle \texttt{ptx}_1, \texttt{ptx}_0 \rangle$: 128-bit plaintext block as a pair of 64-bit elements.
k: 128-bit cipher key
Data: $\texttt{FL}(a, b, c, d)$: non-linear function
Output: $\langle \texttt{ctx}_1, \texttt{ctx}_0 \rangle$: 128-bit ciphertext block as a pair of 64-bit elements

```
1  begin
2      ⟨left, right⟩ ← ⟨ptx_1, ptx_0⟩
3      ⟨k_0, k_1, ..., k_24⟩ ← KeySchedule(k)  // 24 round keys, with 64-bit size
       each
4      left ← left ⊕ k_0                        // note that k_1 is not used
5      for i ← 2 to 23 do
6          if i∉{8, 9, 16, 17} then
7              tmp ←SBOX(left)⊕k_i
8              ⟨tmp_1, tmp_0⟩ ← SplitWord(tmp)      // split up into two 32-bit
               halves
9              tmp_0 ← tmp_0 ⊕ tmp_1
10             tmp_1 ← (tmp_1 ⋙ 8) ⊕ tmp_0
11             tmp ← ⟨tmp_1, tmp_0⟩
12             right ← right⊕tmp
13             ⟨left, right⟩ ← ⟨right, left⟩
14         if i=8  then ⟨left, right⟩ ←FL(left, right, k_8, k_9)
15         if i=16 then ⟨left, right⟩ ←FL(left, right, k_16, k_17)
16     right ← right ⊕ k_24
17     ⟨ctx_1, ctx_0⟩ ← ⟨left, right⟩
18     return ⟨ctx_1, ctx_0⟩
```

Listing A.1: Test executable code

```
#include <stdio.h>
int main(int argc,char[] argv) {
    unsigned char out[BLOCK_SIZE];
    unsigned char key[KEY_SIZE] = {/*Test key*/};
    unsigned char correct[BLOCK_SIZE] = {/*Known good ciphertext*/};
    unsigned char plain[BLOCK_SIZE] = {/*Corresponding plaintext*/};
    int i=1000000;
    if(argc==2) {
        i=atoi(argv[1]);
    }
    for(; i>0; i--) encrypt_block(key,plain,out);
    if(memcmp(out,correct,8)==0)
    {
        printf("OK\n");
        return 0
    }
    else {
        printf("ERROR\n");
        return 1;
    }
}
```

Listing A.2: Source code of the library preloaded into the test executable to measure the number of calls to `rand()`

```
#include <stdio.h>
static unsigned long long counter = 0;
int rand() {
    counter++;
    return 4; // chosen by fair dice roll.
              // guaranteed to be random enough
              // for the purpose of counting calls.
}
void __attribute__ ((destructor)) my_fini(void) {
    printf("\nrand() was called %llu times.\n",counter);
}
```

Listing A.3: Building instructions for the test executable

```
clang -O0 -emit-llvm -S -g input.c
opt -propagametadati input.S -S -o with_metadata.s
opt -sroa -O3 -propagametadati with_metadata.S -S -o optimizedIR.s
#the next step is skipped for unmasked executables
#and replaced by a simple file copy from optimizedIR.s to maskedIR.s
opt -S -ncfa-instruction-replace optimizedIR.s \
  -nocryptofa-security-margin=$SECMARGIN \
  -nocryptofa-mask-everything=$FULLMASK \
  -nocryptofa-masking-order=$ORDER > maskedIR.s
llc -o platform-asm.S maskedIR.s -march=x86-64    #or
llc -o platform-asm.S maskedIR.s -march=arm -mcpu=arm926ej-s
#or
llc -o platform-asm.S maskedIR.s -march=arm -mcpu=cortex-a9
gcc -o executable platform-asm.S #executed on target platform
strip executable  #executed on target platform
```

Bibliography

[1] Giovanni Agosta, Alessandro Barenghi, Massimo Maggi, and Gerardo Pelosi. Compiler-based Side Channel Vulnerability Analysis and Optimized Countermeasures Application. In Charles Alpert, Donatella Sciuto, and Yervant Zorian, editors, *DAC*, pages 77–82. ACM, 2013.

[2] Giovanni Agosta, Marco Bessi, Eugenio Capra, and Chiara Francalanci. Dynamic memoization for energy efficiency in financial applications. In *IGCC*, pages 1–8. IEEE Computer Society, 2011.

[3] Ross J. Anderson, Eli Biham, and Lars R. Knudsen. The Case for Serpent. In *AES Candidate Conference*, pages 349–354, 2000.

[4] ARM Limited. *ARM Architecture Reference Manual*.

[5] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. *ACM Comput. Surv.*, 26(4):345–420, December 1994.

[6] Alessandro Barenghi, Luca Breveglieri, Israel Koren, and David Naccache. Fault injection attacks on cryptographic devices: Theory, practice, and countermeasures. *Proceedings of the IEEE*, 100(11):3056–3076, 2012.

[7] Ali Galip Bayrak, Francesco Regazzoni, Philip Brisk, François-Xavier Standaert, and Paolo Ienne. A first step towards automatic application of power analysis countermeasures. In *Proc. of the 48th Design Automation Conference*, DAC '11, pages 230–235. ACM, 2011.

[8] Suresh Chari, Charanjit S. Jutla, Josyula R. Rao, and Pankaj Rohatgi. Towards Sound Approaches to Counteract Power-Analysis Attacks. In

M. J. Wiener, editor, *CRYPTO*, volume 1666 of *LNCS*, pages 398–412. Springer, 1999.

[9] Jean-Sébastien Coron, Emmanuel Prouff, and Matthieu Rivain. Side Channel Cryptanalysis of a Higher Order Masking Scheme. In P. Paillier and I. Verbauwhede, editors, *CHES*, volume 4727 of *LNCS*, pages 28–44. Springer, 2007.

[10] Blandine Debraize. Efficient and Provably Secure Methods for Switching from Arithmetic to Boolean Masking. In Prouff and Schaumont [21], pages 107–121.

[11] Hiroaki Etoh and Kunikazu Yoda. Propolice: Improved stacksmashing attack detection. *IPSJ SIG Notes*, 75:181–188, 2001.

[12] Ulrich Greveler, Benjamin Justus, and Dennis Loehr. Forensic content detection through power consumption. In *ICC*, pages 6759–6763. IEEE, 2012.

[13] Yuval Ishai, Amit Sahai, and David Wagner. Private Circuits: Securing Hardware against Probing Attacks. In D. Boneh, editor, *CRYPTO*, volume 2729 of *LNCS*, pages 463–481. Springer, 2003.

[14] Uday Khedker, Amitabha Sanyal, and Bageshri Karkare. *Data Flow Analysis: Theory and Practice*. CRC Press, Inc., Boca Raton, FL, USA, 1st edition, 2009.

[15] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential Power Analysis. In *Proc. of*, CRYPTO '99, pages 388–397. Springer, 1999.

[16] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86. IEEE, 2004.

[17] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power Analysis Attacks: Revealing the Secrets of Smart Cards (Advances in Information Security)*. Springer, 2007.

[18] Andrew Moss, Elisabeth Oswald, Dan Page, and Michael Tunstall. Compiler assisted masking. In Prouff and Schaumont [21], pages 58–75.

[19] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[20] NIST. FIPS-46-3: Data Encryption Standard (DES). `http://www.itl.nist.gov/fipspubs/`, October 1999.

[21] E. Prouff and P. Schaumont, editors. *Cryptographic Hardware and Embedded Systems - CHES 2012. Proc.*, volume 7428 of *LNCS*. Springer, 2012.

[22] Matthieu Rivain and Emmanuel Prouff. Provably Secure Higher-Order Masking of AES. In S. Mangard F-X Standaert, editor, *CHES*, volume 6225 of *LNCS*, pages 413–427. Springer, 2010.

[23] Kai Schramm and Christof Paar. Higher order masking of the AES. In *Proc. of*, CT-RSA'06, pages 208–225. Springer, 2006.

[24] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. Addresssanitizer: A fast address sanity checker. In *USENIX ATC 2012*, 2012.

[25] Claude E. Shannon. Communication theory of secrecy systems. *The Bell System Technical Journal*, 28(4):656–715, October 1949. A footnote on the initial page says: "The material in this paper appeared in a confidential report, 'A Mathematical Theory of Cryptography', dated Sept. 1, 1945, which has now been declassified.".

[26] Michele Tartara and Stefano Crespi-Reghizzi. Continuous learning of compiler heuristics. *TACO*, 9(4):46, 2013.

[27] I.A. Zabotin, G.P. Glazkov, and V.B. Isaeva. Cryptographic Protection for Information Processing Systems: Cryptographic Transformation Algorithm. Technical Report GOST 28147-89, 1989. (translated by A. Malchik, with editorial and typographic assistance of W. Diffie).