



**POLITECNICO**  
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE  
E DELL'INFORMAZIONE

EXECUTIVE SUMMARY OF THE THESIS

## Compiler assisted modeling of side channel information leakage

LAUREA MAGISTRALE IN COMPUTER SCIENCE AND ENGINEERING - INGEGNERIA INFORMATICA

**Author:** FRANCESCO GHIANDA

**Advisor:** PROF. GIOVANNI AGOSTA

**Co-advisor:** ALESSANDRO BARENGHI

**Academic year:** 2023-2024

### 1. Introduction

Nowadays, in our daily lives, we are surrounded, even without realizing it, by many *embedded* devices that store and share people's data, some of which are particularly sensitive, e.g. payment cards or identification cards, only to cite two. To protect the data from an unintended access, these devices contains specialized hardware to execute a cryptographic algorithm and to store the cipher key.

Unfortunately, even if a *computationally secure* cryptographic algorithm was used, the device may still be vulnerable to the so called side-channel attacks (SCA). I propose a tool that dynamically model the power consumption of device to help the identification of vulnerable ciphers from SCAs.

#### 1.1. Side channel attacks

SCAs exploit the dependency between the information leaked from the device during the execution of the cipher, such as the power consumption or the electromagnetic emission, and the data processed in the same time instant [5], to gain information about the secret key and eventually reconstruct it.

To understand how these attacks work, it's necessary to first understand how these phys-

ical properties of the device are correlated to the value of the cryptographic key stored inside it. The circuits composing the device are built using CMOS (Complementary Metal-Oxide-Semiconductor) logic cells. The power consumption of the device is the sum of the power consumption of all the CMOS logic cells composing it and the one of a single logic cell is the sum of two contributes: The static power consumption  $P_{stat}$  and the dynamic power consumption  $P_{dyn}$  [11]. The one that we are interested in, is the dynamic power consumption, because it is the power consumed when the logic cell switches, i.e. when the logic value goes from 0 to 1 or vice versa, and so it depends on the data processed by the device.

**Power models** To model the power consumption of a device on the data it is processing, two mathematical models can be used:

- **The Hamming weight model**, which is the simplest one, assumes that the instantaneous power consumption of a circuit is proportional to the number of bits set in the value the circuit is currently processing [11].
- **The Hamming distance model**, that is an evolution of the previous one, estimates

the power consumption of the device, counting how many transitions (switches in the circuit) occur [11].

The Hamming distance needs two values, that are consecutively written in the same register, in order to be computed. Taken two values  $v_1$  and  $v_2$ , their Hamming distance can be computed as the Hamming weight of the value obtained xoring  $v_1$  and  $v_2$ .

$$HD(v_0, v_1) = HW(v_1 \oplus v_2)$$

**Passive side channel attacks** SCAs can be categorized into two subtypes: the passive SCAs and the active SCAs.

The passive SCAs are the ones in which the attacker does not interfere with the execution of the algorithm but he limits himself to measuring the physical quantities required to carry out the attack [11].

The two main passive SCAs techniques are:

- Simple power analysis (SPA) [6, 9, 15]
- Differential power analysis (DPA) [6, 9, 15]

**Simple power analysis** To conduct a power analysis attack, the attacker first has to measure the power consumption of the device during the execution of cipher. The set of measurements composes a power trace [11].

In the SPA only one or few power traces are collected and directly interpreted [8]. This methods can be useful to understand the major characteristics of the device, but carry out a full attack can be quite challenging and may require the use of complex statistical methods [11].

**Differential power analysis** With the DPA, described by Kocher et al. in [8], a large number of traces is required to reduce the noise and to reveal small variation in the power consumption. The general approach to conduct a DPA attack is described in [11] and is composed of three steps:

- **Power consumption hypothesis:** In the first step the attacker chooses a point in the cipher algorithm, typically in the first or in the last round, and using a power model, simulates the power consumption of the device for every possible key hypothesis.
- **Power traces collection:** The attacker can now measure the real power consump-

tion of the device, collecting all the power traces he needs.

- **Correlation:** The attacker can now correlate the real measurements with the hypothetical ones to find the correct secret key.

**Active side channel attacks** In an active SCA, the attacker tampers the device, e.g. altering the power supply or generating an abnormal clock signal [7], to obtain an abnormal behavior. The difference between the results collected during the normal condition and the tampered one, can be exploited to gain information about the secret key [11].

## 1.2. Automated analysis

The first step to protect the device against SCAs is to understand which are the most vulnerable part of the cipher. The countermeasures that can be applied come with significative overhead and so they cannot be applied to all the instructions of the cipher, knowing that we are considering embedded devices with limited capabilities.

Even if this analysis can be done manually, it can be long, complex and prone to errors. This is why a compiler-based technique is preferable.

**Security oriented data flow analysis** Introduced by Agosta et al. in [2], Security oriented Data Flow Analysis (SDFA) is a technique based on the classical data flow analysis, aimed at quantify the vulnerability of each intermediate value of the cipher to an SCA, computing the dependency of the intermediate values from the secret key.

The SDFA was successfully implemented in the tool described by Sanfilippo in [14], that represents the starting point of the work presented here. The tool besides identifying the vulnerable instructions, automatically applies countermeasures with a threshold approach. For each instruction a vulnerability index is calculated, and to the instructions that have this index below the threshold, the countermeasure is applied.

## 1.3. Countermeasures

Countermeasures against SCAs can be implemented in hardware or in software. Here we examine the latter, which are the less expensive ones, and can be applied to any already existing

implementation.

The two main countermeasures that can be automatically applied are called *masking* and *hiding* [11]. The goal of the countermeasures is to remove the dependencies that exist between the processed data and the power consumption of the device, making the correlation impossible.

**Masking** The masking countermeasure works by modifying the cipher’s instructions, masking each vulnerable intermediate value with one or more random values, called *masks* that changes at each execution. These random values *share* the secret and increase the number of measurements needed to carry out the attack.

The number  $d$  of *masks* used is known as the order of masking, and it’s proven that only a  $(d + 1)$ -th-order attack can break a  $d$ -th-order masking [3]. In practice, given that the difficulty of carry out a  $d$ -th-order attack increases exponentially with  $d$ , only few implementations of order greater than 1 exist [2].

**Hiding** The hiding countermeasure, unlike the masking, does not modify the cipher’s instructions, but it tries to achieve the goal interfering with the execution. There are two ways to implement hiding, and can be both applied together:

- **Random insertion of dummy operation:** At each execution different dummy operations are randomly inserted between the cipher’s ones, which will be executed at a different time instant on every execution [11].
- **Shuffling:** Reschedule randomly the instructions of the ciphers at each execution, keeping the final result correct [11].

## 2. Compilers

As stated in the previous section, compilers play a fundamental role in code analysis and protection against SCAs.

Compilers are large and complex software, and their modern implementations have a modular design (figure 1) that allows developing each section separately. This design has also another important advantage: suppose that there are  $N$  source languages and  $M$  target machines. If compilers were developed as single monolithic software,  $N * M$  implementations would

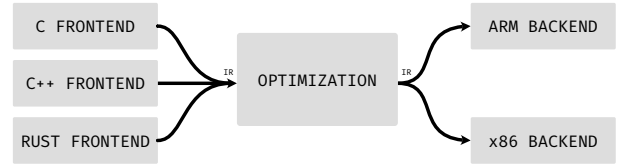


Figure 1: Modular design of a compiler

be needed, instead, the modular design requires only  $N + M$  implementations [4].

During the first phase, lexical, syntax and semantic analyses are performed, and the high-level language is translated into an intermediate representation (IR) that is agnostic to both the source language and to the target architecture. In this way the the analyses performed and the optimizations applied during the second phase can be reused for every combination of source language and target machine.

**Data flow analysis** Data flow analysis (DFA) is a set of static analyses that identify how the data flow between the instructions of a program, and so can be used to identify and quantify the dependency of an instruction of a cipher from the cryptographic key [2]. The DFA works by solving the data flow equations for each node of the control flow graph (CFG). The data flow equations computes two sets of values entering ( $IN[b]$ ) and exiting ( $OUT[b]$ ) a node  $b$ , and are composed by a *transfer function*  $f_s$  that describes how the data are transformed in the node, and a *control-flow constraint* that describes how the data propagate between different nodes. The DFA can be performed in the *forward* direction (from definitions to uses) or in *backward* direction (from uses to definitions). The equations 1a and 1b are the general data flow equation of a node  $b$ , respectively for the forward and backward analysis.

$$\begin{cases} OUT[b] = f_s(IN[b]) \\ IN[b] = \bigcup_{\forall p \in pred(b)} OUT[p] \end{cases} \quad (1a)$$

$$\begin{cases} IN[b] = f_s(OUT[b]) \\ OUT[b] = \bigcup_{\forall s \in succ(b)} IN[s] \end{cases} \quad (1b)$$

**LLVM** LLVM is a compiler *framework* [10] widely used both in research and in production, with which the popular clang compiler was also

created. LLVM follows the compiler modular design (figure 1), and each phase is structured as a pipeline that perform different passes. This structure, together with the wide range of APIs offered by the LLVM libraries, allows to extend the capabilities of the compiler developing passes to perform new analyses or code optimizations, straightforwardly, without modify the already existing ones.

At the core of LLVM there is its intermediate representation known as LLVM IR, which is a first class language with a low level RISC-like instruction set. The code is always in static single assignment (SSA) form, i.e. when each variable is assigned exactly once, and the variables are stored in infinite virtual registers. The code is organized hierarchically in modules which contain global values and functions, and basic blocks that contain the instructions.

### 3. Contribution

The contribution I provided with this work can be divided in two parts. Firstly I rewrote the tool described by Sanfilippo in [14], which was developed using an old version of the LLVM framework, to port it to a more recent version, keeping it up to date and functional. Then, I developed a new set of passes that allow dynamically compute the Hamming distance power model of a device running a cryptographic algorithm, helping to understand which cipher might be vulnerable to an SCA.

**Porting** During this phase, the code of the tool was entirely rewritten as an out of source plugin, fixing the parts that were no longer compatible with the new version of the framework. The main differences between the old and the new version of LLVM are:

- **Pass Manager:** The pass manager (PM) is the LLVM component that organizes the pass pipeline, handling the dependencies between them, and manages the analysis results cache. With the new PM, transformation passes and analysis passes are now managed separately and must be declared differently. Dependencies between different passes are now managed dynamically at runtime, whereas with the old PM they needed to be declared statically.
- **Opaque Pointers:** LLVM used to have

typed pointers in its intermediate representation, but this behavior was changed to simplify the IR and improve some optimizations [1]. The problem was that passes developed using the old version of the framework relied on this information. However, the `load` and `store` instruction still require this type to determine the size of memory to read or write. In fact, in recent version of LLVM, even though pointers have an opaque type, loads and stores are now typed.

The solution is an algorithm that starting from the pointer, recursively travels its uses to find a `load` or `store` operation and retrieves the type from it.

**Power consumption modeling** The new tool developed has the goal of dynamically build a Hamming distance power model of a device for a cipher implementation. To reach the goal, a set of new passes was added to the LLVM framework. These new passes take in input an implementation of a cryptographic algorithm written in a high-level language and produces a library that can then be used to build the power model for that cipher implementation.

The tool is divided in two phases. The first one is performed during the optimization phase of the compiler and so it's independent from the target architecture. The second phase instead needs information about the real physical registers used and so it must be performed only after the register allocation pass, making it target dependent. In this work the version for the ARM architecture was developed.

**Clang annotations** To obtain a better result, new custom clang annotations can be added to the source code of the cipher. These annotations are the attributes `__attribute__((key))` and `__attribute__((plain))` to allow the computation of the instruction dependencies from key and plaintext, and the two pragmas `#pragma cipher round` and `#pragma attack area`. The former is used to identify at which round each instruction belongs, and can be also used to limit the subsequent analyses to a certain number of rounds. The latter can be used to limit the analyses at a particular block of instructions.

**First phase** During the first phase, the instructions that have to be considered are identified, and for each instruction a use-def chain, i.e. a chain of instructions that starting from the inputs of the program reach the considered instruction, is computed. Using the computed use-def chains, for each instruction a new function that computes its intermediate value is emitted. The metadata containing the key and plaintext dependencies information are added to each instruction to be later used.

**Memory dependencies handling** When an implementation contains data structures, for example to store the subkeys, that are translated using `alloca` instructions in the IR, i.e. when the data structures were kept in memory, this results in the presence of the so called *memory bridges*. The majority of these memory bridges are removed by LLVM during the *mem2reg*, which tries to transform memory operations into register operations. However, the *mem2reg* pass is not always able to remove all of them. When this happens, an instruction can have a memory dependency with other instructions that do not belong to its use-def chain, and if not taken in consideration, this can lead to incorrect result. A specialized pass handles this problem by identifying every possible memory dependency for each instruction.

**Size optimization** Emitting a function for each instruction, which contains all the instructions needed to compute the intermediate value from the start of the program, can eventually lead to a very big binary that cannot be used on embedded devices with limited flash memory. An optimization can be enabled to significantly reduce the size of the emitted code. The optimization causes the emitted functions to no longer contain all the instructions of the use-def chain, but only the last instruction, which computes the final value, and calls the already emitted functions for each instruction operand. The drawback of this optimization is that it slows down the execution due to the many function calls that must be executed.

**Second phase** The second phase exploits the emitted functions that compute the intermediate values, in order to emit new functions that

will compute the Hamming distance (HD). Finally a header file containing all the definitions for the emitted HD functions, is produced to help the usage of the library.

In order to emit a Hamming distance function, two information is needed:

- Two machine instructions that write on the same register one after the other.
- A mapping between these two instructions and two of the previously emitted functions.

The first information is obtained scanning all the machine instructions in execution order, and creating a map between each register and the instructions that writes on it.

The second information is more complex, since there isn't a direct way in LLVM to link an IR instruction to a MIR instruction. A pass tries to create this mapping using the metadata of the debug location that is present either on IR and MIR instructions.

## 4. Experimental results

The tool was tested on a STM32F411E-DISCO board, featuring an ARMv7 32-bit Cortex-M4 CPU running at 72 MHz, 512 Kbytes of flash memory and 128 Kbytes of RAM [?].

The collected data was used to create, for each cipher analyzed, a plot of the value Hamming distance in execution order, and is colored to show the dependencies of the instructions from the key and/or the plaintext.

Eleven ciphers plus some variants were tested. The implementations of the Triple-DES (TDES2 and TDES3) have produced a binary file too large to be tested on the board used, even with the optimization enabled, while the implementations of CAST and Serpent, do not terminate the execution.

The results also shown that the mapping between the IR instructions and the MIR instructions in real implementations is not perfect because the translation from IR to machine code changes the instructions too much in some cases, e.g. in cases of register spilling.

**AES result** AES (Rijndael) was developed in 1997 and became the Advanced Encryption Standard in 2002 [11]. It is structured as a Substitution-Permutation Network, and elaborates blocks of 128 bits. The key can be 128, 192 or 256 bits long, which also determines the

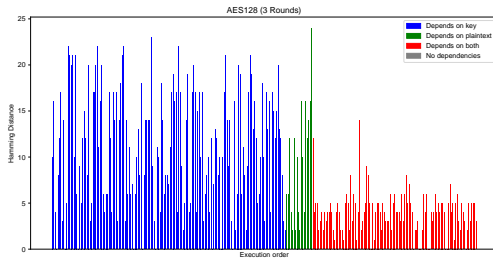


Figure 2: Plot generated for the AES variant with 128 bits key

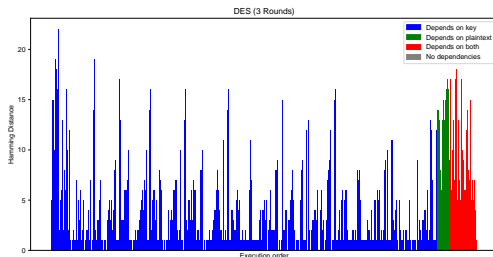


Figure 3: Plot generated for DES

number of rounds, that are respectively 10, 12 or 14 [13]. Three rounds of each variant of the cipher were tested. In figure 2 is reported the plot built with the data of the 128 bits key variant. In the plot is easy to see three different sections: in blue the key schedule, in green the loading of the plaintext in the state, and in red the three rounds analyzed.

**DES result** Another interesting case is DES. The algorithm was the old standard until it was replaced by AES. It has a Feistel network structure that perform 16 rounds on 64 bit blocks with a key of 56 bits [12]. The plot in figure 3 shows how the key schedule of DES is very large compared to the rounds. Is also possible to observe several repeated peaks in the key schedule part, that represent the loading the different key's portions. Even in this case the first 3 rounds of the cipher were analyzed.

## 5. Conclusions

The work produced is the result of extensive research in the fields of cryptography and compilers. The analyses performed using the developed tool has yielded results compatible with the already known characteristics of different cipher

implementations. I hope that the work done can be a contribution to the research in this field, and help develop new solutions for protecting devices from side channel attacks.

## 6. Acknowledgements

First and foremost, I would like to express my deepest gratitude to my advisor Giovanni Agosta and my co-advisor Alessandro Barenghi. I really appreciate the support you gave me throughout the course of this work and the precious knowledge you shared with me. You have passed on to me the passion you have for this field, which I will certainly treasure in my future years.

I am also grateful for the luck I had in being able to study in such a high-level institution that is the Politecnico di Milano. The years spent here have shaped me not only as a student but also as a person.

Here I met those who became some of the most important people to me. Thank you Josselyn and thank you Silvia for your sincere friendship. Ali, you are like a brother to me. I am extremely grateful to you for always being there.

Last but not least, I would like to thank from the bottom of my heart my parents Elena and Danilo, my brother Lorenzo, my aunt Barbara and my grandparents Cesira, Augusto and Elda. Without you none of this would have been possible.

## References

- [1] Opaque Pointers — LLVM 20.0.0git documentation.
- [2] Giovanni Agosta, Alessandro Barenghi, Massimo Maggi, and Gerardo Pelosi. Compiler-based side channel vulnerability analysis and optimized countermeasures application. In *Proceedings of the 50th Annual Design Automation Conference*, pages 1–6, Austin Texas, May 2013. ACM.
- [3] Giovanni Agosta, Alessandro Barenghi, and Gerardo Pelosi. Automated instantiation of side-channel attacks countermeasures for software cipher implementations. In *Proceedings of the ACM International Conference on Computing Frontiers*, pages 455–460, Como Italy, May 2016. ACM.

- [4] Andrew W Appel and Jens Palsberg. *Modern Compiler Implementation in Java, Second Edition*.
- [5] Hasindu Gamaarachchi and Harsha Ganegoda. Power Analysis Based Side Channel Attack, January 2018. arXiv:1801.00932 [cs].
- [6] Himanshu Gupta, Subhash Mondal, Rana Majumdar, Neha Sana Ghosh, Soumya Suvra Khan, Ngala Etienne Kwanyu, and Ved P Mishra. Impact of Side Channel Attack in Information Security. In *2019 International Conference on Computational Intelligence and Knowledge Economy (IC-CIKE)*, pages 291–295, Dubai, United Arab Emirates, December 2019. IEEE.
- [7] Dusko Karaklajic, Jorn-Marc Schmidt, and Ingrid Verbauwhede. Hardware Designer’s Guide to Fault Attacks. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 21(12):2295–2306, December 2013.
- [8] Paul Kocher. Differential Power Analysis. 1999.
- [9] Paul Kocher, Joshua Jaffe, Benjamin Jun, and Pankaj Rohatgi. Introduction to differential power analysis. *Journal of Cryptographic Engineering*, 1(1):5–27, April 2011.
- [10] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86, San Jose, CA, USA, 2004. IEEE.
- [11] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power analysis attacks: revealing the secrets of smart cards*. Springer, New York, 2007. OCLC: ocm71541637.
- [12] National Institute of Standards and Technology (NIST). Data Encryption Standard. January 1977.
- [13] National Institute of Standards and Technology (NIST), Morris J. Dworkin, Elaine Barker, James R. Nechvatal, James Foti, Lawrence E. Bassham, E. Roback, and James F. Dray Jr. Advanced Encryption Standard (AES). *NIST*, November 2001. Last Modified: 2024-07-25T12:07:04:00 Publisher: National Institute of Standards and Technology (NIST), Morris J. Dworkin, Elaine Barker, James R. Nechvatal, James Foti, Lawrence E. Bassham, E. Roback, James F. Dray Jr.
- [14] Stefano Sanfilippo. *A compiler-based technique for automated analysis and protection against side-channel attacks*. PhD thesis, Politecnico di Milano, 2016.
- [15] Raphael Spreitzer, Veelasha Moonsamy, Thomas Korak, and Stefan Mangard. Systematic Classification of Side-Channel Attacks: A Case Study for Mobile Devices. *IEEE Communications Surveys & Tutorials*, 20(1):465–488, 2018.