POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Compiler assisted modeling of side channel information leakage

Tesi di Laurea Magistrale in
Computer Science and Engineering - Ingegneria Informatica

Author: **Francesco Ghianda**

Student ID: 974688
Advisor: Prof. Giovanni Agosta
Co-advisors: Alessandro Barenghi
Academic Year: 2023-24

Abstract

This thesis describes a software developed with the goal of facilitating the protection of block ciphers against *side-channel* attacks. Side-channel attacks can be used to reconstruct the cryptographic key by exploiting information such as the power consumption or the electromagnetic emission of the device executing the cipher, compiled with statistical techniques. These attacks are particularly effective against embedded devices and can be carried on even with algorithms considered computationally secure from classical attacks.
The software was implemented using the LLVM framework, developing new passes in different compilation phases, in order to obtain, for each cipher, a library of functions that allow dynamic modeling of the device power consumption based on the Hamming distance model.
Finally, 11 ciphers were analyzed, some with multiple variants, and the graphs constructed from the obtained data highlighted known characteristics of the different cryptographic algorithms.
The thesis also analyzes the different types of side-channel attacks and the possible countermeasures for each of them, emphasizing how compilers are an essential tool to facilitate, reduce possible errors and make the application of countermeasures more efficient and precise.

Keywords: block ciphers, LLVM, compilers, side-channel attacks, power analysis, automated analysis and countermeasure application

Abstract in lingua italiana

In questa tesi viene descritto un software che è stato sviluppato con l'obiettivo di facilitare la protezione dei cifrari a blocchi dagli attacchi *side-channel*. Gli attacchi side-channel sono in grado di ricostruire la chiave crittografica sfruttando informazioni come il consumo di potenza o l'emissione elettromagnetica del dispositivo che esegue il cifrario, in combinazione con tecniche statistiche. Questi attacchi sono particolarmente efficaci su dispositivi embedded e funzionano anche su algoritmi considerati computazionalmente sicuri agli attacchi classici.
Il software è stato implementato usando il framework LLVM, sviluppando dei nuovi passi in diverse fasi della compilazione così da ottenere per ogni cifrario una libreria di funzioni che permettono di modellare dinamicamente il consumo di potenza del dispositivo basandosi sul modello della distanza di Hamming.
Infine sono stati analizzati 11 cifrari, alcuni con più varianti, e i grafici costruiti dai dati ottenuti hanno messo in evidenza caratteristiche note dei diversi algoritmi crittografici.
Nella tesi si analizzano anche quali sono i diversi tipi di attacchi side-channel e le possibili contromisure per ciascuno di essi, enfatizzando come i compilatori siano uno strumento essenziale per facilitare, ridurre i possibili errori e rendere più efficiente e precisa l'applicazione delle contromisure.

Parole chiave: cifrari a blocchi, LLVM, compilatori, attacchi a canale laterale, analisi di potenza, analisi e applicazione contromisure automatizzata

Contents

Abstract **i**

Abstract in lingua italiana **iii**

Contents **v**

Introduction **1**

1 State of the art **3**

1.1 Cryptosystems . . . 3

1.1.1 Cryptographic devices . . . 4

1.2 Side channel attacks . . . 4

1.2.1 Passive side channel attacks . . . 5

1.2.2 Active side channel attacks . . . 6

1.3 Power analysis . . . 8

1.3.1 Power consumption . . . 8

1.3.2 Power model . . . 9

1.3.3 Simple Power Analysis . . . 11

1.3.4 Differential power analysis . . . 12

1.4 Countermeasures . . . 14

1.4.1 Countermeasures against fault injection attacks . . . 15

1.4.2 Countermeasures against power analysis attacks . . . 15

1.5 Cryptoshield . . . 18

1.5.1 Frontend extensions . . . 19

1.5.2 Feature recognition . . . 22

1.5.3 Selection . . . 23

1.5.4 Dependency propagation . . . 24

1.5.5 Filtering and Aggregation . . . 25

2 Background **27**
2.1 Compiler . . . 27
2.1.1 Lexical analysis . . . 27
2.1.2 Syntax and semantic analysis . . . 29
2.1.3 Intermediate representation . . . 31
2.1.4 Code generation . . . 32
2.2 Control flow graph . . . 33
2.2.1 Dominance relation . . . 35
2.3 Static single assignment . . . 36
2.3.1 Construction of the SSA form . . . 38
2.4 Data Flow Analysis . . . 39
2.4.1 Data Flow Equations . . . 40
2.4.2 Reaching Definitions . . . 40
2.4.3 Liveness Analysis . . . 41
2.5 LLVM . . . 42
2.5.1 LLVM IR . . . 44
2.5.2 LLVM APIs . . . 49

3 Software architecture **53**
3.1 Cryptoshield porting . . . 53
3.1.1 Pass Manager . . . 54
3.1.2 Opaque Pointers . . . 57
3.1.3 Cipher rounds annotation patch . . . 59
3.2 Dynamic Analysis . . . 60
3.2.1 Clang Annotations . . . 62
3.2.2 Middle-End Passes . . . 62
3.2.3 Backend Passes . . . 70

4 Validation **75**
4.1 Testing Environment . . . 76
4.2 Ciphers Analyzed . . . 76
4.2.1 AES . . . 78
4.2.2 DES . . . 78
4.2.3 CAST . . . 79
4.2.4 CLEFIA . . . 79
4.2.5 HIGHT . . . 79
4.2.6 MISTY1 . . . 80
4.2.7 SEED . . . 80

4.2.8 Serpent 80
4.2.9 Simon and Speck 80
4.2.10 XTEA 81
4.3 Results 81

5 Conclusions **91**

Bibliography **93**

List of Figures **99**

List of Tables **101**

List of Algorithms **103**

Acknowledgements **105**

Introduction

In an era where almost all personal data is stored and processed digitally, cryptography and in particular the use of secure cryptographic algorithms has become essential to protect people's privacy and authenticate the information. This aspect does not only regard devices that can directly access internet, such as personal computers or smartphone, that are more exposed to external threats and where the use of cryptography is evident, but also many other contexts that pervade everyday life, such as transit passes, SIMs, payment cards, electronic identity cards, and many others. All these devices are **smart cards**, that are *embedded* devices with dedicated hardware to execute a cryptographic algorithm and store the secret key, which must never be revealed, under penalty of compromising the device itself, which could have serious consequences.
The advancement of cryptography studies has let in recent decades to increasingly secure and efficient ciphers, but often this security is evaluated only by considering the mathematical properties of an algorithm.
However, there is a relatively new category of attacks called side-channel attacks (SCA). This type of attack exploits additional information, such as the power consumption, the time variation or the electromagnetic emission, that leaks from the device during the execution of the cryptographic algorithm. This information is correlated to the control flow of the algorithm and to the data the device is processing, and can be measured by an attacker, who then, using statistical analysis, can reconstruct the secret key. SCAs are particularly effective against embedded devices.

In this thesis, the main techniques with which SCAs are carried out, and the possible countermeasures that can be adopted to increase the resistance of ciphers to these types of attacks are analyzed, with particular focus to SCAs that exploit the power consumption of an embedded device to reconstruct the key of a block cipher. It is also shown how, using compilers, it is possible to implement automated analyses and countermeasure applications on cipher implementations, significantly reducing the time required and the possible errors that can be made with a manual approach.
Compilers are generally seen as software that simply translates a high-level language, such as C, to the machine code that can be executed, but in reality, they are very complex

software divided in several parts, which in addition to translating the code, perform a series of analyses and code transformations to apply optimizations that allow programs to be executed efficiently.
Besides that, compilers have a large responsibility in the security of programs, as they can modify the code, e.g. by inserting stack canaries to prevent stack overflow or by enabling the address space layout randomization (ASLR) which makes more difficult to execute arbitrary code.

Subsequently, a tool based on the LLVM framework is described in detail. The tool exploits the possibility offered by the framework, to add a set of *static* analyses performed on the intermediate representation of the cipher implementation, that quantifies the vulnerability of each instruction to SCAs, and automatically apply countermeasures.
The tool was developed using an old version of LLVM, and the first part of the work done in this thesis was the porting of the tool to a recent version of the framework.
After that, the updated tool was used as a starting point to develop new LLVM passes composing a new tool, which taking as input a cipher implementation, it produces a library containing new functions that yields different information for each instruction of the cryptographic algorithm, such as possible dependencies from plaintext and key, and more importantly the Hamming distance between two intermediate values written consecutively in the same CPU register. This last information is strictly correlated to the power consumption of the device and can be used to build a model of its power consumption.
The tool, therefore, allows the *dynamic* analysis of a cipher implementation to understand what the vulnerable points to a SCA attack may be.

1 State of the art

When considering the security provided by a cryptographic algorithm, it's common to think that a cipher is secure if there are no way to break its mathematical structure, or at least if a possible theoretical attack is practically infeasible by the means of time or requires too much computational resources to be carried out successfully. In fact, an algorithm is considered secure if the fastest way to broke it is by a brute-force attack.

However, there are some information that can be used by an attacker to bypass the security of a cipher and gain possession of the secret key.
This information is physical properties of the device that is executing the cipher, such as the power consumption or the electromagnetic emission, that are dependent on the control flow of the algorithm and on the data it is elaborating, and especially in embedded device can leak and can be measured by the attacker. The leak of one of these properties is called a side-channel, in contrast to the main-channel of information that flow in the cryptographic algorithm as intended, and the attacks based on this information are called side-channel attacks (SCA).

In this chapter is explained what a side-channel attack is, the main techniques used to carry out an SCA and what the possible countermeasures are to protect ciphers implementations against these types of attacks, providing a view on where the research on this filed is arrived, with particular interest in methods that offer an automated way of analysis and protection of the code.
The last section of the chapter describes "Cryptoshield", a tool developed leveraging the LLVM framework, that can be used to analyze and apply countermeasure to ciphers implementations in an automated way, and which represents the ground point to the work presented in this thesis in chapter 3.

1.1. Cryptosystems

A cryptosystem is defined as a pair of cryptographic methods for encrypting and decrypting data. The primary aim of a cryptographic method is to provide confidentiality,

integrity, and authenticity of the data.
The two cryptographic methods are encryption and decryption and can be represented by the two mathematical functions c = E(p, ke) and p = D(c, kd), where p is the plaintext, c is the ciphertext, and ke and kd are, respectively, the cryptographic keys for encryption and decryption.
In modern cryptography, cryptosystems are designed following the Kerckhoffs's principle which, in contrast with the practice of security through obscurity, states that "the security of a cryptosystem must lie in the choice of its keys only; everything else (including the algorithm itself) should be considered public knowledge" [35].
For an attacker, breaking a cryptographic algorithm means finding the cryptographic key to gain access to the sensible protected data. Most of the cryptographic algorithms are vulnerable to brute-force attack [38], in which an attacker, in possession of a ciphertext, tries every possible combination of the key, to find the right one. Even if that type of attack is theoretically possible, modern algorithms use keys 128 or 256 bits long, which means that there are 2^{128} or 2^{256} possible keys to try out. Even using a modern computer this attack is infeasible.
If an attacker can find the cryptographic key in less time than it would take him to use a brute-force attack, then the cryptographic algorithm is broken.
A cryptographic algorithm is considered computationally secure if breaking it requires an amount of computing power that is not available in practice [30].

1.1.1. Cryptographic devices

Cryptographic devices are hardware devices such as smart cards that store the cryptographic key and execute a cryptographic algorithm. The advantage of these devices, unlike classic computers, is that they are not connected to the Internet, and it is not possible to install additional software that could compromise them [30].

1.2. Side channel attacks

When a cryptographic device is performing a cryptographic operation (encryption or decryption), beside the intended output (the plaintext or the ciphertext), it might leak some other unintended information [40]. For example when a device is executing some code, it consumes an amount of power that can be measured and that is correlated with the control flow of the program and the elaborated data.
SCAs (side-channel attacks) target this types of vulnerabilities in hardware or software implementations of cryptographic devices, rather than trying to directly break the math-

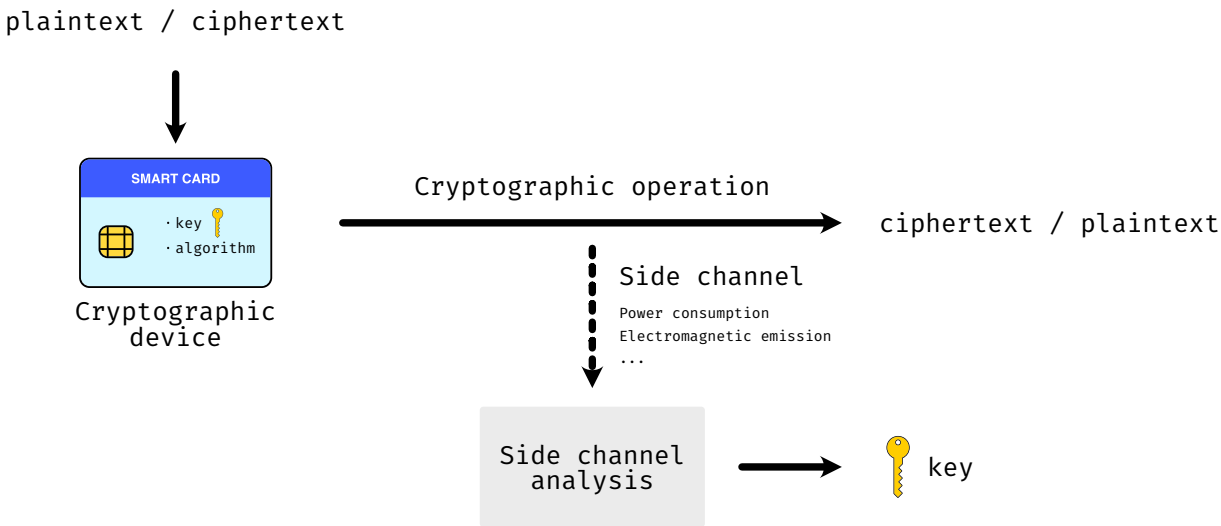


Figure 1.1: General representation of a side channel attack

ematical structure of the algorithm; even if an algorithm is considered computationally secure, it may be vulnerable to side-channel attacks.
Attackers measure physical quantities such as power consumption, electromagnetic emission, or execution time that are related to what is being executed internally in the system, and exploit relationships between these measurements and the inputs in order to discover the secret key.
The cryptographic device and its algorithm can be viewed as a black box, because in order to execute an SCA, the attacker does not require a deep understanding of the implementation of the attacked device or the cryptographic algorithm that it executes. The only information needed is the algorithm implemented in the device, but this information is usually of public domain [16].

1.2.1. Passive side channel attacks

In passive SCAs the attacker does not interfere with the operation of the cryptographic device, but observes its behavior while performing cryptographic operations. The secret key is revealed by observing the physical properties of the device [30].
Common passive side channel attacks techniques are:

- Simple power analysis (SPA) [23, 27, 40]
- Differential power analysis (DPA) [23, 27, 40]
- Correlation power analysis (CPA) [27]
- High-Order DPA [27]

- Template attacks [27]
- Simple electromagnetic attacks (SEMA) and differential electromagnetic attacks (DEMA) [27]
- Timing attacks [23]
- Acoustic attacks [23]

Later in this chapter, SPA and DPA will be described in detail, as this thesis is mainly focus on attacks that exploit the power consumption of the device, and these two techniques are the main ones. CPA and High-Order DPA are two variants of the DPA [27].

1.2.2. Active side channel attacks

In an active attack, the cryptographic device, its inputs and/or its environment are manipulated to make the device behave abnormally, or to bypass implemented security mechanisms. The secret key is revealed by exploiting the so obtained device's abnormal behavior [30].
Karaklajic et al. in [25] listed the most common techniques that an attacker can use to tamper the device.
Power supply alteration: Under-powering or producing power spikes during the execution of a cryptographic operation can cause memory faults. This technique is simple and low cost, but it's difficult to control the timing of the fault introduction.
Clock glitches: If a device requires an external clock generator to work (e.g. smart cards), the attacker can generate an abnormal clock signal that can cause the execution of unordered instructions or store in memory invalid data. The technique is one of the simplest but cannot be applied to devices that has an internal clock generator.
Temperature attacks: Exposing the cryptographic device to excessively high or low temperature can lead to memory corruption.
Electromagnetic fault injection: An external electromagnetic field can cause the abnormal switch of a transistor resulting in a modification of the memory content.
Optical attacks: As semiconductors are sensible to laser ionization, exposing a decapsulated chip to a very strong light source can cause the switch of a transistor. Using focused laser beam a single and precise bit in memory can be set or reset.
Some of the possible attacks that can be performed are: [25, 36, 40]

- Differential fault analysis (DFA)
- Tampering the program flow

- Safe-errors attacks

Differential fault analysis (DFA)

To accomplish a differential fault analysis attack, the attacker has to exploit the differential information between correct and faulty ciphertexts to retrieve the secret key.
First the attacker collects several ciphertexts with the cryptographic device operating in normal condition, then collects other ciphertexts produced from the same device with the same inputs (plaintext and secret key), but this time tempering the execution of the cryptographic operation with one of the techniques described above.
When the attacker have collected sufficient faulty and faulty-free ciphertexts, he applies differential cryptanalysis to construct and then solve differential equations and obtain bits of the secret key [25].
Pierre Dusart et al. in [19], explains that this type of attacks was firstly used in 1996 by Boneh et al. to find cryptographic keys of public key cryptosystems (e.g. RSA). The attack has later been extended by Biham and Shamir to various secret key cryptosystems such as DES. The authors then show how this type of attack can be used against AES, the current encryption standard.

Tampering the program flow

In this type of attack, the attacker, instead of tampering the data, alters the program counter of the processor, causing instructions to be skipped or executed multiple times. In this way the attacker is able to leak some sensible data, reduce the number of rounds of a symmetric cipher or even bypass security countermeasures that check if the device was tampered at the end of the execution [25].

Safe-errors attacks

While in DFA the attacker finds the secret key analyzing the correlation between the tamper-free and the tampered outputs of the cryptographic operation, in a safe-error attack, the attacker analyzes the fact that an error introduced during the execution of the cryptographic operation may or may not produce an observable effect on the output. The fact that in this type of attack, the attacker is not interested in the output itself, allows it to be used on a cryptographic device with countermeasures that validate the result [25]; in fact, if a countermeasure yields an error due to an invalid output, the attacker knows that the injected error has produced an effect on the result.

1.3. Power analysis

Power analysis is a type of passive side channel attack that exploits the dependency between the power consumed by a cryptographic device and the data processed and/or the operation performed by the device at the same instant [22].
To conduct a power analysis attack, the attacker, first has to measure the power consumed by the cryptographic device. This is done by inserting a shunt resistor in series with the power supply or the ground of the cryptographic device. Then, with an oscilloscope, he can measure the voltage drop and obtain the power consumed by the device, for every instant, during the execution of the cryptographic operation. This set of measurements composes a power trace [30].
Analyzing a single or multiple power traces the attacker is able to reconstruct the secret key.

1.3.1. Power consumption

Cryptographic devices are built using CMOS (Complementary Metal-Oxide-Semiconductor) circuits. The power consumption of a CMOS circuit is the sum of the power consumption of the logic cells composing the circuit.
A CMOS logic cell is made up using an NMOS (N-channel Metal-Oxide-Semiconductor) and a PMOS (P-channel Metal-Oxide-Semiconductor) transistors that work in a complementary mode. When the pull-up network (PMOS) is active, the pull-down network (NMOS) is inactive, so the GND and the V_{DD} are never directly connected, significantly reducing the current flow and thus the power consumed by the circuit.
For example the figure 1.2 represents the circuit diagram of a CMOS inverter. When the input signal is high (1) the NMOS circuit (N1) is active and the output is connected to GND (0). When the input signal is low (0), the PMOS circuit (P1) is active and the output is connected to V_{DD} (1).
The power consumption of a CMOS logic cell can be divided into two components: The static power consumption P_{stat}, and the dynamic power consumption P_{dyn}.

Static power consumption is the power consumed by the circuit when the input signal is constant. It's derived by the leakage of current flowing through the transistor (I_{leak}) currently inactive, and typically it is very small [30].

$$P_{\text{stat}} = I_{\text{leak}} \cdot V_{\text{DD}} \tag{1.1}$$

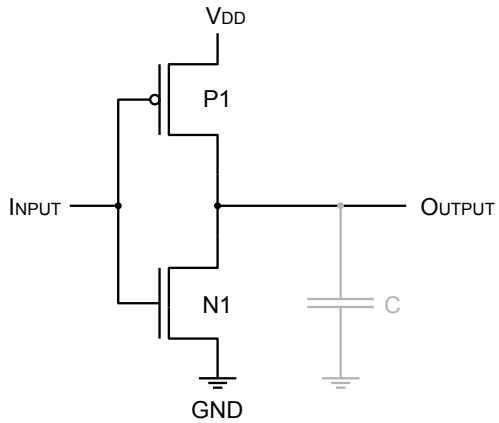


Figure 1.2: Circuit diagram of a CMOS inverter

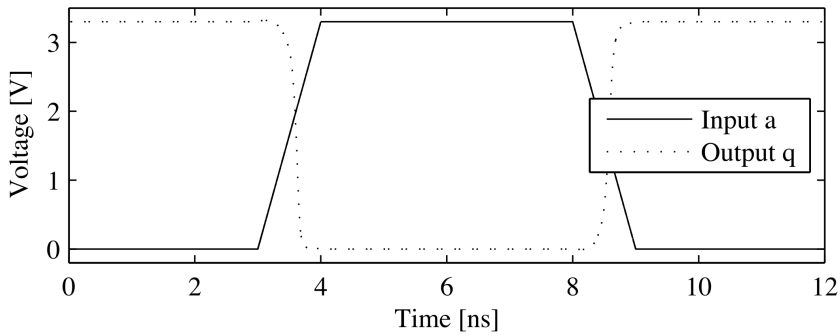
Dynamic power consumption is the power consumed by the circuit when the logic cell switches and so depends on the data that is processed by the circuit. For example, performing an *exclusive or* operation on two numbers with higher Hamming distance will consume more power than performing the same operation on two numbers with a lower Hamming distance [27, 30].
Typically, dynamic power consumption is much higher than the static one, because when a logic cell switches, the capacitance of the cell needs to be charged and for a short period of time a short circuit between GND and V_{DD} in the CMOS is observed [30]. Figure 1.3a shows the input and output signals from a CMOS inverter. The input signal changes two times, firstly from 0 to 1 and then back from 1 to 0. The second graph (1.3b) represents the current draw by the circuit, where the two times when the CMOS switch and consume more power are clearly visible.

1.3.2. Power model

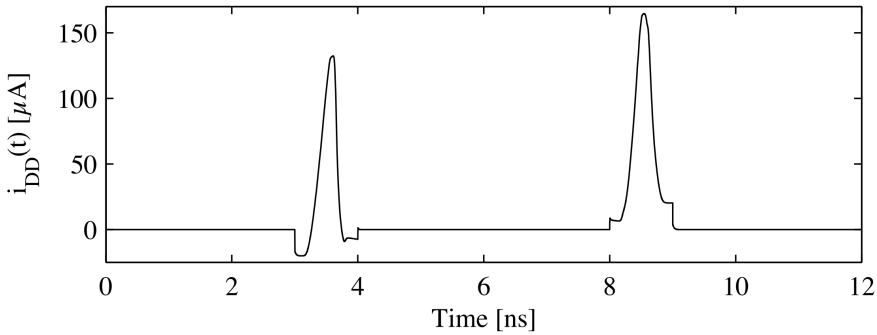
A power model is a mathematical model used to predict or estimate the power consumption of an electronic circuit. Typically, power models are used during the design process of a digital circuit to determine if the circuit meets the specification, but an attacker can use it to simulate the power consumption of a device for a given plain-text and secret key, and compare the result with the actual power consumption of the device [30].

Hamming Distance power model

The Hamming distance power model is used to count the transitions that occur in a circuit. Since most of the power is consumed when there is a switch in the transistors of the circuit, knowing how many transitions occur can be used to estimate the power



(a) Input and output signal of a CMOS inverter



(b) Instantaneous current drawn by the CMOS inverter

Figure 1.3: Dynamic power consumption of a CMOS inverter [30]

consumption [22, 30].
The Hamming distance of two values v_0 and v_1 can be defined as the Hamming weight of the value obtained xoring v_0 with v_1.

$$\mathrm{HD}(v_0, v_1) = \mathrm{HW}(v_0 \oplus v_1) \tag{1.2}$$

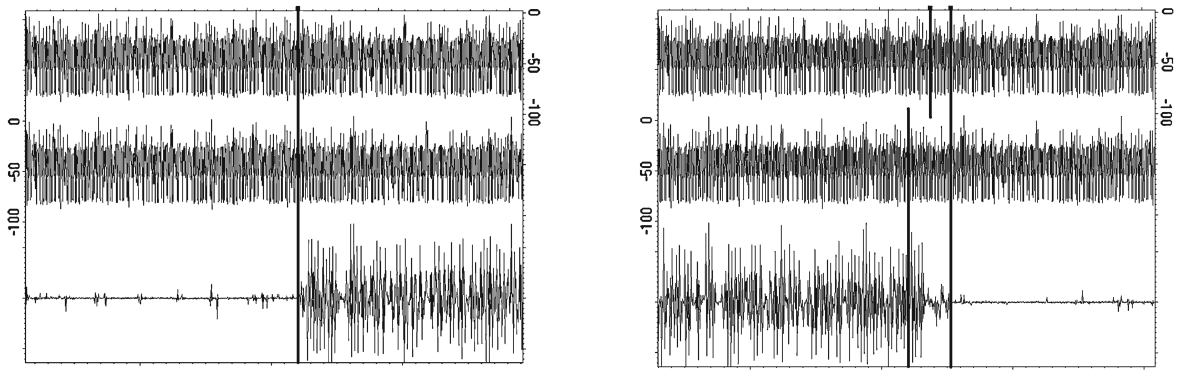
The Hamming weight (HW) is simply the number of bits set to one in a value, so the Hamming distance is the number of bits that differ between v_0 and v_1.
Since for the calculation of the Hamming distance it is necessary to have two values, this model assumes that the attacker knows both the value contained in the register before the switch and the value after it.
Another important aspect to consider is that "the Hamming distance model assumes that all cells contribute to the power consumption equally and there is no difference between $0 \rightarrow 1$ and $1 \rightarrow 0$ transitions" [30].

Hamming Weight power model

Hamming weight model is a simpler model used when the attacker does not know two consecutive values for the same register, and so he cannot compute the Hamming distance to estimate the power consumption. In the Hamming weight model, the assumption is that the power consumption is proportional to the number of bits set in the current value of the data that is being processed [30].

1.3.3. Simple Power Analysis

Kocher et al. in [26] defines simple power analysis (SPA) as "a technique that involves directly interpreting power consumption measurements collected during cryptographic operations. SPA can yield information about a device's operation as well as key material", which means that SPA can be used when only one or few power traces are available.
In practice, carrying out an attack with SPA can be challenging and requires a detailed knowledge of the cryptographic algorithm under attack and, especially, with only one power trace, the use of complex statistical methods to extract the signal from the noise [30].
Single trace analysis: When a single power trace is available, a visual inspection of the trace can reveal information about the data-dependent branches and which institutions are performed based on their execution time. Algorithms with data-dependent branches are particularly vulnerable to SPA, but even in cryptographic algorithms without data-dependent branches (e.g. AES) some instructions power consumption can have variations based on the data processed [11, 27].
Multi trace analysis: Multiple power traces can be recorded using the same plaintext multiple times or changing the plaintext for every power trace. In the first case, the collected power traces can be used to reduce the noise by computing a new power trace that is the mean of the recorded ones [30].
In the other case, different power traces can be compared to find differences that are data-dependent [27]. For example in figure 1.4a the two upper traces were recorded using two different plaintexts and the same key. The third trace is obtained computing the difference between the two recorded traces. It can be noted that in the first part the two traces are very similar, as their difference remains flat, but at some point, the traces diverge, which means that the computations took different paths. In the second figure 1.4b, the second trace has been shifted by 21 clock cycles, and now the traces are synchronized where they weren't before. This means that the two computations have taken two different paths and the path taken by the second computation took 20 more clock cycles than the first one's.



(a) (b)

Figure 1.4: Two power traces and their difference [27]

1.3.4. Differential power analysis

Firstly described by Kocher et al. in [26], DPA is one of the most important and used technique to reveal the secret key of a cryptographic device exploiting its power consumption. The popularity of this type of attack is due to the fact that it does not require in-depth knowledge of the cryptographic device under attack, but only of the cryptographic algorithm used.

Unlike SPA, which mainly exploit the instruction flow dependency, DPA reveals how the power consumed a device depends on the data processed. For this reason, DPA attacks works better on algorithms, such as AES, that do not have data-driven control flow.

As the data-dependent variations in power consumption are relatively small, compared to those derived on a change in the control flow [27], DPA requires, to be effective, a large number of power traces to reduce the noise present in the measurements as much as possible.

All DPA attacks follow a general approach [30] shown schematically in figure 1.5.

Power consumption hypotheses Firstly, the attacker chooses an intermediate point in the cryptographic operation where it is easier to carry out the attack. For example, in AES attacking the first round, just after the substitution with the S-Box is quite simple, because the intermediate result depends only on one byte of the secret key.

The attacker generates a set of subkeys hypotheses, and knowing the cryptographic algorithm that the device executes, using a power model, such as the Hamming distance power model, described in section 1.3.2, he can simulate the power the device will consume executing the cryptographic algorithm, for a set of different known inputs. In the table 1.1 there is a simple example of power consumption simulation with only 3 different inputs and every possible subkey. The intermediate result is the output of the substitution box

(S-Box) of AES (table 1.2), taking in input the xor between the input and the subkey hypothesis.

Input	Subkey hypothesis	Intermediate result	Hamming distance
0xF5	0x00	0xCF	4
0x38	0x00	0xEC	4
0x4E	0x00	0x69	4
...	...	...	...
0xF5	0xFF	0xE0	3
0x38	0xFF	0x10	2
0x4E	0xFF	0xAF	4

Table 1.1: Power consumption simulation.

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**A**	**B**	**C**	**D**	**E**	**F**
0	63	7C	77	7B	F2	6B	6F	C5	30	01	67	2B	FE	D7	AB	76
1	CA	82	C9	7D	FA	59	47	F0	AD	D4	A2	AF	9C	A4	72	C0
2	B7	FD	93	26	36	3F	F7	CC	34	A5	E5	F1	71	D8	31	15
3	04	C7	23	C3	18	96	05	9A	07	12	80	E2	EB	27	B2	75
4	09	83	2C	1A	1B	6E	5A	A0	52	3B	D6	B3	29	E3	2F	84
5	53	D1	00	ED	20	FC	B1	5B	6A	CB	BE	39	4A	4C	58	CF
6	D0	EF	AA	FB	43	4D	33	85	45	F9	02	7F	50	3C	9F	A8
7	51	A3	40	8F	92	9D	38	F5	BC	B6	DA	21	10	FF	F3	D2
8	CD	0C	13	EC	5F	97	44	17	C4	A7	7E	3D	64	5D	19	73
9	60	81	4F	DC	22	2A	90	88	46	EE	B8	14	DE	5E	0B	DB
A	E0	32	3A	0A	49	06	24	5C	C2	D3	AC	62	91	95	E4	79
B	E7	C8	37	6D	8D	D5	4E	A9	6C	56	F4	EA	65	7A	AE	08
C	BA	78	25	2E	1C	A6	B4	C6	E8	DD	74	1F	4B	BD	8B	8A
D	70	3E	B5	66	48	03	F6	0E	61	35	57	B9	86	C1	1D	9E
E	E1	F8	98	11	69	D9	8E	94	9B	1E	87	E9	CE	55	28	DF
F	8C	A1	89	0D	BF	E6	42	68	41	99	2D	0F	B0	54	BB	16

Table 1.2: AES S-Box Table

Power traces collection At the same time, the attacker measures the real power consumed by the device during the cryptographic operation, using the same set of inputs used to generate the hypothetical power traces. The collected traces have to be time synchronized and this requirement could be achieved using a trigger signal to start the cryptographic operation, sent by measuring equipment [36].
Collecting a large number of traces can help reduce noise, making the analysis more precise.

Correlation Once the attacker has collected enough power traces from the device, he can now compare the hypothetical power traces with the real ones, finding, presumably, one

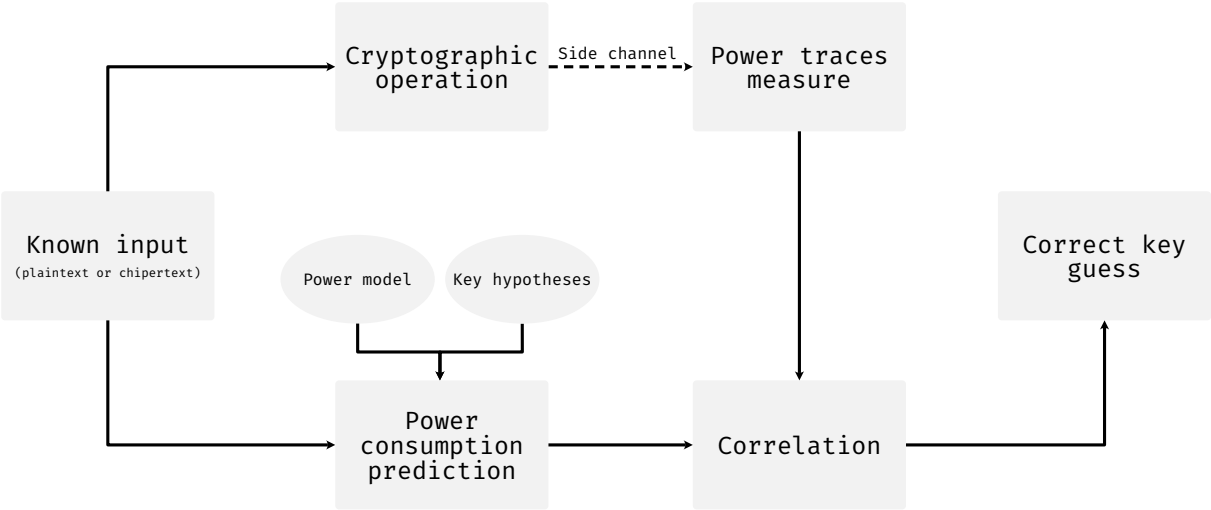


Figure 1.5: Differential power analysis attack scheme

set of hypothetical power traces associated with a subkey, which is better correlated with a set of real power traces. In that case the hypothetical secret subkey used to generate this power trace is equal to the real secret subkey contained in the cryptographic device.

These steps can be repeated multiple times to discover other portions of the secret key, until the entire key is reconstructed.

1.4. Countermeasures

Side channel attacks exploit information leaked from a cryptographic device, measuring physical properties that are dependent on the data that the device is elaborating or with the control flow of the cryptographic algorithm. To protect cryptographic devices against side channel attacks, the hardware or the software needs to be modified to reduce the information leaked, or make them unusable to reconstruct the correct secret key.
The advantage of hardware techniques is that they are transparent to the algorithm the device executes, making them more resistant to bugs in the software, but are typically expensive and must be added during the design phase of the cryptographic device, making them impractical in some cases. Software countermeasures are instead more simple to implement and their use can restore the security in hardware-protected system where the underlying hardware protections have been compromised [6].

1.4.1. Countermeasures against fault injection attacks

This type of countermeasure are not the focus of this work but are briefly explained as they can be applied together with other countermeasures to achieve a better protection of the cryptographic device. Below are presented two techniques found in literature, but many other techniques exist to protect the devices from different types of attacks.

Passive and active shields Passive shields are metal layers that cover a part or the entire chip, preventing optical fault injection or probing attacks. Active shields consist of a metal mesh that transmits signals across the surface of the chip and detects any interruptions on a wire [25].
Duplication and comparison With this method, some hardware blocks are duplicated and, at the end, there is a comparator that checks that the computation has not been tampered with [23].

These countermeasures can be used to physically reduce the quantities that can be measured, for example, with logic gates that have smaller electromagnetic emission, or to make device tampering more difficult.

1.4.2. Countermeasures against power analysis attacks

The main goal of countermeasures is to alter the dependency between the information leaked and the computation of the algorithm.
The two main techniques that can be applied are called *masking* and *hiding* [30]. Both masking and hiding try to break the link between the processed data and the power consumption of the device, but the former modify directly the computation of the algorithm, instead the latter leave the computation unaltered.
To achieve a better protection, different countermeasures can be applied together, but each one involves costs in terms of performance, size of the protected implementation or power consumed by the device. For this reason the best approach is to apply the countermeasures only to the portion of the code that is more vulnerable to side channel attacks.

Automated analysis

Understanding which parts of a cryptographic algorithm can be more vulnerable to an attack, that is, identifying which instructions leak more information through a side-channel, is a fundamental step in effectively protecting an algorithm from this type of attack.
Although this work can be done by visually inspecting the algorithm's assembly instruc-

tions, the complexity and size of cryptographic algorithms make this a difficult and time-consuming task. For this reason, several tools have been developed [6, 8, 9, 20, 29, 36], which, thanks to the use of compilers and already known analysis techniques, help to automate the identification of vulnerable parts of the algorithms and at the same time apply countermeasures.
Typically, these compiler-based techniques statically analyze the code, and using data flow analysis, they figure out which instructions depend on the secret key, which ones depend on the plaintext, and which ones on both, calculating for each instruction a value that indicates its resistance to a possible attack. The instructions that ultimately have a resistance below a chosen threshold are protected using various countermeasures, including masking, hiding, and code morphing.

Security oriented data flow analysis

Introduced by Agosta et al. in [7], Security oriented Data Flow Analysis (SDFA) allows finding out which are the intermediate values that are more exposed to an SCA. The analysis follows the propagation of the key through the intermediate values, and for each bit of each intermediate value, computes the set of secret key bits from which its computation depends.
The more vulnerable intermediate values are the ones that depends only on few bits of the secret key, that typically are located in the first rounds of the cipher. As the computation progress all the key bits are used to compute the intermediate values and the attack becomes more difficult.
SDFA works even in backward as an SCA can be performed also on the ciphertext during the decryption operation, targeting the last rounds. By using SDFA, all the intermediate values are classified based on a vulnerability index which allows the application of countermeasures only to the most vulnerable instructions that can be successfully attacked, reducing the overhead of protecting all the intermediate values.

Masking

Masking is one of the main techniques to protect a cipher implementation from SCAs. This technique works by invalidating the correlation between the power consumption and the data that the dive process [9].
To achieve this, in the protected implementation, every sensitive intermediate value is concealed by one or more random values, called *masks*. The sensitive intermediate values are the ones that depends, directly or indirectly by a portion of the secret key. Masks are generated in the cryptographic device and are different for each execution of the cipher,

so they are not known by the attacker [30].
Considering an intermediate value v, its protected implementation can be written as $v_m = v * m$, where m is the mask and $*$ is an operation chosen between exclusive-or $\oplus$, modular addition $+$ or modular multiplication $\times$, based on the instruction that needs to be masked. The masked intermediate value is represented by a number of shares, that are the masked value v_m and the random values m_i used to conceal the unprotected value. The name *shares* comes from the fact that these values share the secret, and only knowing all the values, the original intermediate value v can be obtained.
The shares of an intermediate value are processed separately and, only at the end of the computation, they are recombined to obtain the result.
The number d of mask used to protect an intermediate value is also known as the order of the masking. It is proven that a d-th-order masking can prevent a d-th-order attack, and can be theoretically broken by a $(d+1)$-th-order attack, i.e. an attack that exploit $d+1$ measurements of the computation of different shared during the same execution [9, 30].
In practice considering that applying several masks for each intermediate value requires more memory and computation time, and because the difficulty of carry out a d-th-order attack increase exponentially with d, only a flew implementation that uses more than two shares ($d > 1$) exists [7, 30].

Hiding

As for the masking techniques, hiding achieves the goal by removing the dependency between the processed data and power consumption. An important difference from the masking technique is that while the latter directly modifies the execution of the cryptographic algorithm, in the hiding technique the computation of the intermediate values remains unchanged.
With hiding the power consumption of the device is randomized acting on the time dimension or on the amplitude dimension [30]. For the time dimension it is important to note that to carry out a DPA attack, all the power traces needs to be aligned in time. There are essentially two ways to interfere with the time dimension:
Random insertion of dummy operation: Inserting dummy operations randomly between the cipher operation, makes the latter appears at different and random point of time on every execution of the algorithm. The number of total dummy instructions inserted must however be always the same, in order to prevent the attacker to gain information about how many dummy instructions are inserted, measuring the time of execution [30]. The insertion of many dummy instructions can cause a slowdown in the execution of the algorithm.

Shuffling: With shuffling, the instructions of the cipher that are independent on each other are rescheduled randomly and differently at each execution. This method, apart from the generation of the random numbers needed for choosing the new order of execution, does not slow down the implementation, but it can only be applied to limited set of instructions that can be reordered without changing the final result [30].
For the amplitude dimension, the idea is to make the power consumption random or equal for all instructions. To achieve this, either the noise needs to be increased or the leaked signal needs to be reduced. There are only a few ways to achieve this behavior in software: [30]

- Using instructions that leaks less information.
- Avoiding having conditional branches dependent on the secret key
- Executing different parallel activities that increase the noise

The level of these protections is quantified by the increase of the number of measurements needed due to the introduced noise. This increase "is shown to be scaling as the square root of the number of unrelated operations performed in the same time instant as the one under attack" [9].

Code morphing

Code morphing is another type of countermeasure where the sensitive instructions are dynamically replaced with other instructions during the execution of the cipher. An example of implementation of code morphing is the MEET approach, proposed by Agosta et al. in [8]. In this approach for every sensitive intermediate value computation, a set of different "code tiles", i.e. code fragments semantically equivalent to the sensitive instruction, is provided. When the cipher is executed, every time there is a sensible instruction, a random code tile for that instruction is chosen and executed instead. A compiler extension modifies the code at compile time, to later achieve this runtime behavior (Figure 1.6). Since the code tile for an instruction is different for each execution, the power traces will also be different.

1.5. Cryptoshield

In this section the Cryptoshield tool is presented, that is the basis of the work done in this thesis. Cryptoshield was firstly developed by Sanfilippo [36], and allows performing security oriented data flow analysis 1.4.2 and automatically apply countermeasures to the most vulnerable instructions. The tool is built as an extension of the LLVM framework and

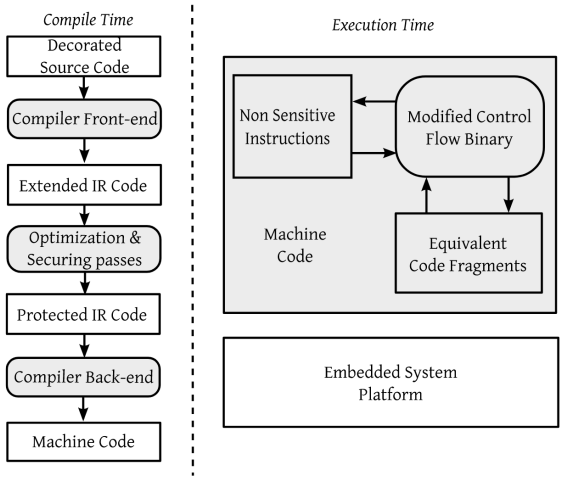


Figure 1.6: The architecture of the MEET approach [8].

includes a collection of analyses organized in different passes which, from an unprotected high level source code (e.g. C), produces a protected binary.
The passes of the analysis process can be divided in two different macro categories each one subdivided on more specialized blocks that contain the single passes (Figure 1.7). The first one is the **vulnerability analysis**:

- Feature recognition
- Selection
- Forward and backward propagation
- Filtering and aggregation

The second one is the **countermeasure application**:

- Masks application

1.5.1. Frontend extensions

Before the SDFA can start, the high level source code needs to be prepared with the addition of annotations and the execution of LLVM passes in order to produce an intermediate representation (IR) with some characteristics that helps to simplify the subsequent analysis.

Loop unrolling One of the first steps in the analysis is checking that the control flow graph (CFG) is loop-free. This condition is required because after this pass, the analysis

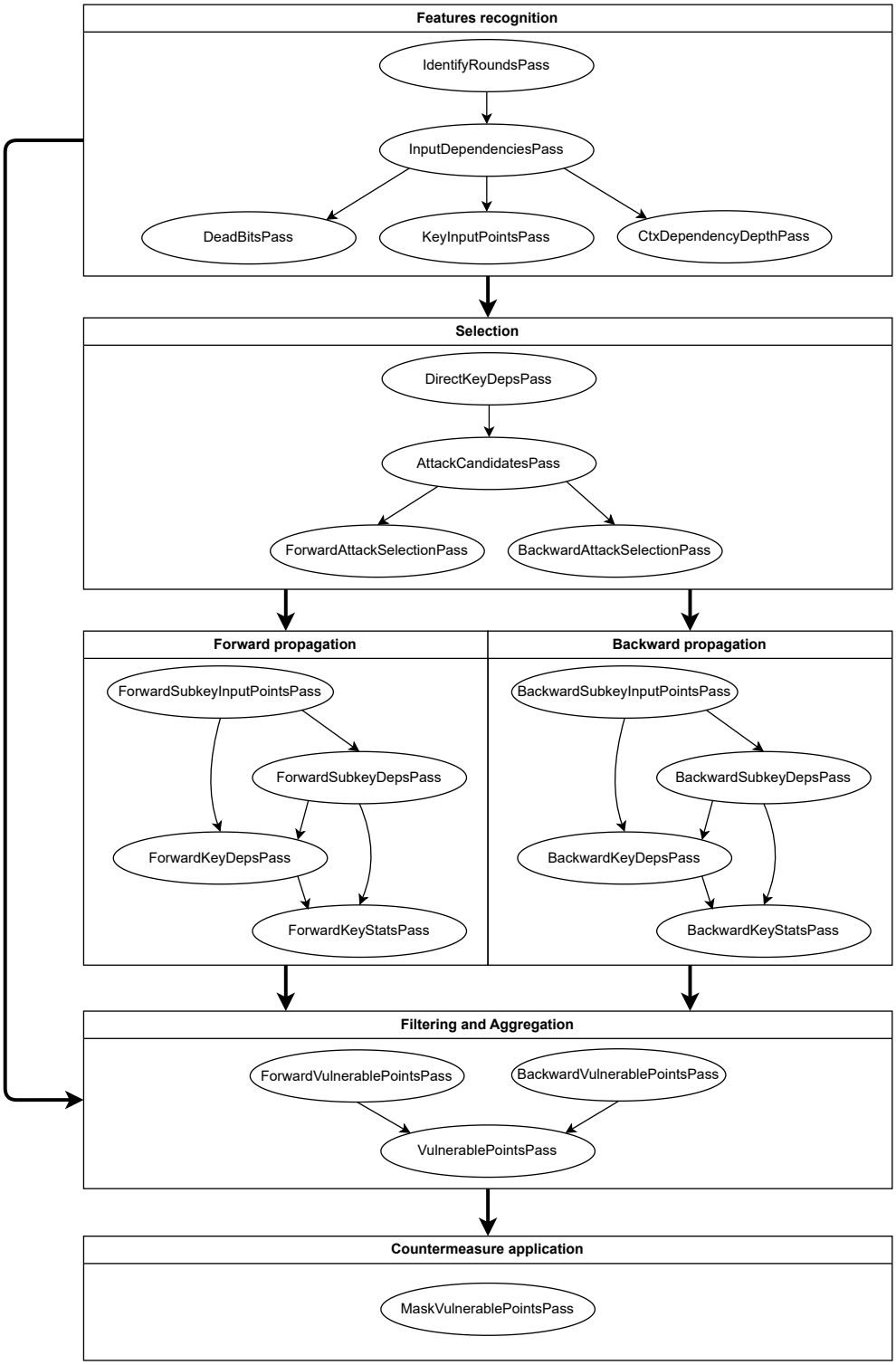


Figure 1.7: Organization of the LLVM passes implemented and the visualization of the decency between them [36]

take for granted that an instruction has only a unique value associated. In order to have a loop-free CFG the loops in the source code must be annotated with `#pragma unroll` that will be later handled by clang. The boundaries of a loop that needs to be unrolled must be known at compile time, but this requirement is not particularly limiting, since all ciphers have loops with static indices.

Cipher variable markers To identify which are the inputs of the secret key and of the plaintext, the parameters of the cryptographic function must be annotated with two new attributes that are `__attribute__((key))` and `__attribute__((plaintext))`. Clang was modified to handle thees attributes and generate two intrinsics functions, `llvm.crypto.key` and `llvm.crypto.plain`, in the IR, that respectively takes as single parameter the variable that contains the secret key and the plaintext. The functions are defined opaques, i.e. the optimizer does not know their implementation, and so they cannot be removed by an optimization pass.

S-Box markers Similarly to the markers for key and plaintext, possible S-Boxes must be marked with the new attribute `__attribute__((sbox))`. Clang will handle this attribute by producing another intrinsic function called `llvm.crypto.sbox`.

Rounds delimiters To help the debug of the analysis and to produce the final visualizations an optional marker can be introduced in the source code, to delimit the rounds of the cipher. This is done by wrapping the code of the round in a new scope (i.e. in C a block delimited with braces) and marking that scope with `#pragma cipher round`. This new pragma will be handled by inserting the IR of the round in a basic block delimited with two loops containing only the two intrinsics `llvm.crypto.begin` and `llvm.crypto.end`. An analysis pass will later re-identify the instructions belonging to the different rounds, and the loop introduced will be removed.

An example of annotated cipher is presented in Listing 1.1 with the toy cipher used by Agosta et al. in [7]. Only the `__attribute__((sbox))` is missing from this example.

Listing 1.1: Example of annotated cipher

```
#include "stdint.h"

#define rot32(x,n) (((x) << (n)) | ((x) >> (32-(n))))

inline __attribute__((always_inline))
void keyschedule(uint32_t *subkeys, const uint32_t *key) {
    subkeys[0] = key[1];
    subkeys[1] = key[0];
    subkeys[2] = key[0] ^ key[1];
    subkeys[3] = key[1] << 10 ^ 0xFCEF;
}

void crypt(uint32_t __attribute__((key)) *key,
    const uint32_t __attribute__((plain)) *input,
    uint32_t *output) {

    uint32_t subkeys [4]; uint32_t tmp = *input;
    keyschedule(subkeys, key);

    #pragma unroll
    for (int n = 0; n < 4; ++n) {
        #pragma cipher round
        {
            tmp ^= subkeys[n];
            tmp = rot32(tmp, 2+n);
        }
    }

    *output = tmp;
}
```

1.5.2. Feature recognition

Feature recognition is the first phase of the analysis and it is in charge of identifying the properties of the implementation of the cryptographic algorithm, such as rounds, s-boxes and key dependency. Particularly, the first pass, **IdentifyRoundsPass**, exploit the annotations introduced before, to identify the instructions that belong to a round of the cipher.

InputDependencyPass uses the *key* and *plaintext* attributes to identify the variables containing the secret key and the plaintext. The pass, exploring the Data Flow Graph

(DFG), also calculates the dependency depth for the key and the plaintext, that is, for each instruction, the minimum distance in the DFG from the input point to the considered instruction, and the ciphertext output point, which is the input of the **CtxDependencyDepthPass** that computes the dependency depth for the reverse DFG.

DeadBitsPass identifies possible s-boxes exploiting the appropriate attribute, and for each of them, computes the *dead bits*, i.e. bits of the output of the s-box that remain fixed at 0 or 1 independently of the input, and so they do not carry useful information.

Finally, **KeyInputPointsPass** assigns to each instruction that was previously identified as a *key input point*, the subrange of key bits that the instruction is loading. The dimension of this subrange is equal to the size of the instruction's result. The pass ensures to assign all the subranges in order to cover all the keys, keeping track of the number of total key bits assigned.

1.5.3. Selection

In the selection phase the instructions that represent a possible point to mount an SCA are identified. The analysis assumes that the instructions identified in this phase represents a lower bound on the implementation security and therefore there is no attack that can be carried out at lesser or equal cost by targeting other instructions outside those identified in this phase.
Firstly **DirectKeyDepsPass**, starting from the key input points identified in InputDependencyPass, propagates the dependencies to all the instructions, assigning to each of them a vector of bits. Each bit of this vector is set if the instruction depends on the corresponding key bit.

AttackCandidatesPass search the instructions that are more easy to attack. In a symmetric block cipher the instructions in the outermost rounds are more vulnerable, because only few bits of the key are mixed with the plaintext and so the relation with the measured quantities is simpler. This pass looks for the instructions that depends on a portion of the key but not on the plaintext, and that transitively depends on portion of the plaintext. The pass finally computes two mappings for the forward and backward analysis that map the found attack candidates with their dependency depth calculated before.

AttackPointsSelectionPass extract two subsets, for forward and backward analysis, from the attack candidates found in the previous set. The *attack points* are selected in order to construct a set of instructions that cover all the key bits with the smallest dependency depth possible. Finally, the pass builds the *subkey to key matrix* whose

columns represent the bits of the subkey of the attack points, and the rows represent the bits of the key. Each cell of the matrix is set to 1 if the bit of the subkey (column) depends on the correspondent bit of the key (row).

1.5.4. Dependency propagation

In the dependency propagation phase, the information retrieved in the selection phase is propagated to all the instructions of the implementation to finally obtain a vulnerability index for each instruction. This phase is "repeated" two times, one for the forward direction and one for the backward direction.
The first step, **SubkeyInputPointPass**, assigns and initializes a pair of *vulnerability matrices* to each key input point. The matrices called L, for linear dependencies, and NL, for non-linear dependencies, have a row for each bit of the key, and a column for each bit of the instruction result. The cells of L and NL are set to 1 if the bit of the instruction result depends respectively linearly and non-linearly on the correspondent bit of the key. The diagonal of the L matrices of the instructions identified as attack candidates, is initialized with correspondent vector of key bits.

SubkeyDepsPass propagates the dependencies information to all the instructions, following specific rules for linear and non-linear matrices, based on the propagation direction (forward or backward) and on the type of the instruction, i.e. linear boolean operation (`XOR`), non-linear boolean operation (`OR`, `AND`), arithmetic operation (`ADD`, `SUB` and `MUL`, `DIV`), shifts operations, truncations or extensions and table lookups.

In the **KeyDepsPass**, the *vulnerability matrices* just calculated and the *subkey to key matrices* obtained in AttackPointsSelectionPass, are used together to produce the *mediated key dependency matrices*, that indicates for each instruction, which bit of the key influences each bit of an intermediate result. This is a fundamental information because it is directly correlated with the vulnerability of the instruction. The last step, **KeyStatsPass**, calculates the *vulnerability index* for each instruction. The vulnerability index is calculated for each bit of the instruction result, as the minimum bit of the key, the bit of the result depends on. The vulnerability index for the instruction is the minimum of the vulnerability indices of the bits.
The pass computes, for each instruction, 4 different vulnerability indices based on 4 different attack models. The overall vulnerability index is the minimum among the four. Smaller vulnerability index indicates more vulnerable instruction, apart from the special case of vulnerability index equal to 0 that indicates that the instruction does not depend on any key bit and so is not vulnerable.

1.5.5. Filtering and Aggregation

The last phase of the analysis identifies the instructions that need to be protected. **ForwardVulnerablePointsPass** and **BackwardVulnerablePointsPass** identify vulnerable instructions, whose vulnerability indices are below a specified threshold. Finally, **VulnerablePointsPass** collect the two sets of vulnerable instructions in a single set ready for countermeasure application.
The vulnerable instructions are selected following two rules:

- **Forward rule**: An instruction is selected if its vulnerability index is below the threshold and one of its operands is already marked vulnerable or is part of the key schedule.
- **Backward rule**: An instruction is selected if its vulnerability index is below the threshold and is a direct user of a subkey fragment or one of its uses depends on the plaintext and is marked vulnerable.

The threshold can be chosen to maximize protection or by making a compromise between protection and performance. A good value for the threshold is the dimension of the key, for which in only the outermost rounds (more vulnerable) of the cipher the countermeasures is applied.

2 Background

In this chapter the fundamental concepts on how a compiler works are explained and in particular a large section is dedicated to LLVM, the compiler framework used to develop the work presented in this thesis. Compilers are an essential tool for building **automated** analysis and countermeasure application systems against side channel attacks. Before these types of tools were developed, the protection of each cipher implementation was left to the programmer that had to analyze the assembly code manually to discover some vulnerabilities. As one can imagine, this work was very long, complex and prone to errors.

2.1. Compiler

A typical static compiler is a complex software whose main function is to translate *source code* written in high-level language, in *assembly code* or *machine code* for a specific architecture. Modern compilers, including LLVM, are usually developed in a modular way, dividing the compilation steps in 3 different modules (Figure 2.1).
The advantage of this approach is that each frontend module can be developed independently of the target machine and each backend module can be developed without having to consider all the characteristics of the high-level language. Another important aspect is that if there are N high-level language and M different target machines, developing a modular compiler requires only $N + M$ frontend and backend modules, instead of $N * M$ monolithic compilers [14].
A more detailed view of all the phases of a compiler is represented in Figure 2.2. The first 4 phases corresponds to the frontend of the compiler, while the last two corresponds to the backend. The machine independent optimization phase is optional [10].

2.1.1. Lexical analysis

The lexical analysis phase takes as input the high-level language source code, and produces a collection of tokens. A *token* is a pair composed of a *token-name* and a possible *token-attribute*, that is passed to the parser (syntax analyzer) when requested. To create a token, the lexer (or scanner) reads the characters of the source code and groups them

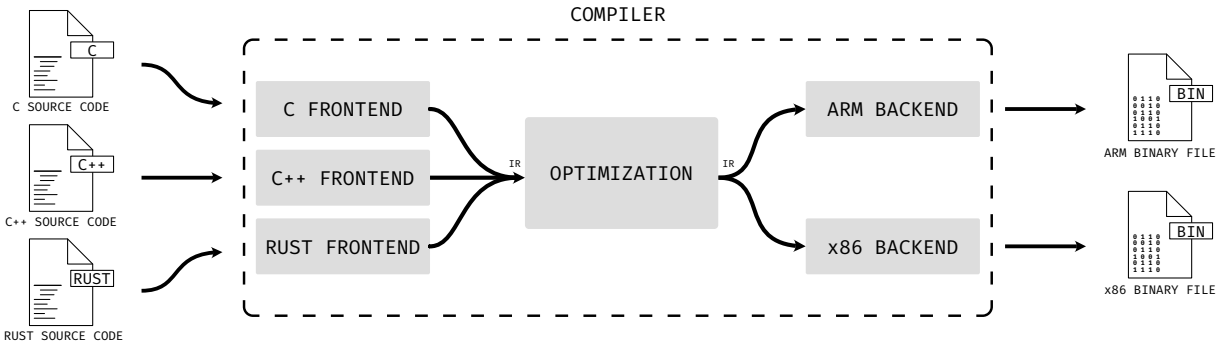


Figure 2.1: High level representation of a modular compiler

in sequences called *lexemes*. Lexemes are matched with a *pattern* and transformed in the correspondent token. Typically, the parser is the component that requires a new token to the lexer when needed [10].

For example the fragment `x = x + y * 2` of C source code, can be tokenized as following:

- **x** is an identifier and is translated to the token ⟨ID, x⟩
- = is the assign symbol translated to the token ⟨=⟩
- **y** as for x, y is another identifier and becomes ⟨ID, y⟩
- ***** is an operator and is mapped to the token ⟨*⟩
- **2** is a constant integer so becomes ⟨CONST, 2⟩

Notice that some tokens, associated with keywords and operators, do not need an attribute. In a real compiler the names, types and positions of identifiers are maintained in a symbol table and in the token attribute there is the pointer to the correct entry of the table [10]. So the overall code fragment is translated in the stream of tokens:

⟨ID, 1⟩ ⟨=⟩ ⟨ID, 1⟩ ⟨+⟩ ⟨ID, 2⟩ ⟨*⟩ ⟨CONST, 2⟩

Index	**Identifier**	
1	x	...
2	y	...

Table 2.1: Symbol table

A lexer, that is able, reading the characters of the source code, to classify them in different tokens, is developed starting from a *regular expression*, i.e. a sequence of symbols that

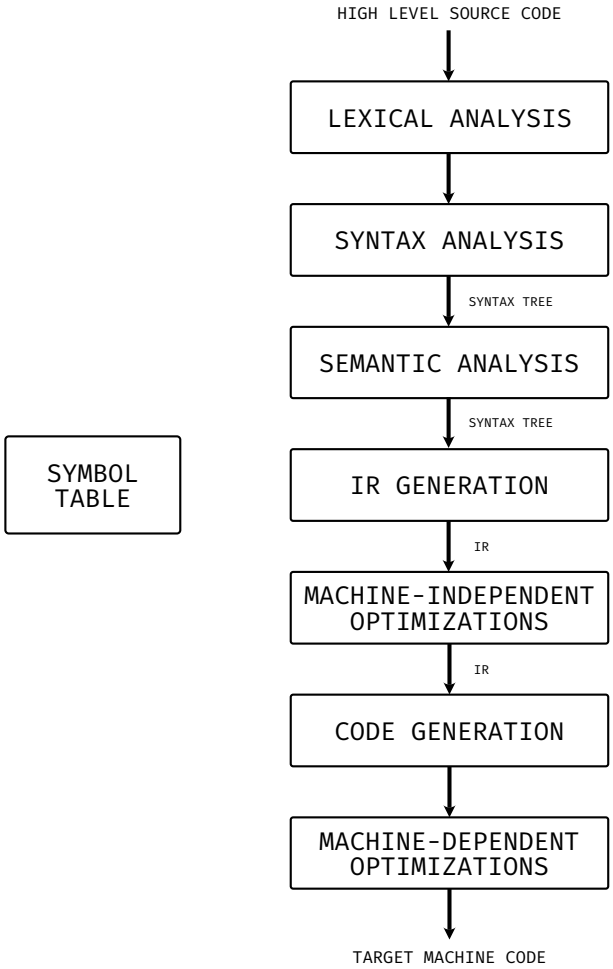


Figure 2.2: Detailed view of the phases of a compiler [10].

defines rules to identify a pattern. The regular expression is an easy way to express a language, but it's difficult to translate it in a software. To do that, the regular expression is converted firstly in a non-deterministic finite automaton and then in a deterministic finite automaton (DFA). At this point the conversion of the DFA in the program that compose the lexer is straightforward. Usually this work is not done by hand, but there are lexer generators that take as input the definitions of the tokens, as regular expressions, and generate the correspondent lexer [32].

2.1.2. Syntax and semantic analysis

During the syntax analysis phase, also called *parser*, the tokens produced by the lexer are used to produce a *syntax tree*. As the name suggests, a syntax tree is a tree-like structure where each interior node represents an operation and the children of that node represent the argument of the operation [10]. This phase, unlike the lexer, works in a less local way, so it is able to find out if the program under compilation is not well-formed and emit a

syntax error [32].
A parser can be defined by a *context-free grammar*, which defines the rules of derivation to build the syntax tree. Even if in theory a parser that recognize all the context-free grammars can be built, in practice, for performance reason, only LL and LR parser exists [10].
Both LL and LR parsers read the input from left to right, but while the LL parsers construct the syntax tree in a top-down way (from the root to the leaves), LR parsers construct the tree in a bottom-up way.

Context-free grammars

To build a parser, the regular expression used during the lexical analysis are not enough powerful. A regular expression (or finite automata) can only recognize regular languages, but they can not recognize a context-free language. For example with a regular expression it is impossible to tell if the parenthesis in an expression are balanced or not. A context-free grammar is defined as a quadruple $\mathcal{G}(N, T, P, S)$ where:

- N is a finite set of *non-terminal* symbols, i.e. a symbol of the grammar that can be expanded when a production is derived.
- T is a set finite set of symbols, that are the symbols of the alphabet of the language the grammar generates. In this context, the terminal symbols are all the possible token-names recognized by the lexer.
- P is a set of productions. A production is a rule of the grammar expressed in the form:

  $$\textit{left side} \rightarrow \textit{right side}$$

  The left side of a production contains only a non-terminal symbol, while the right side contains a string of non-terminal and terminal symbols.
- $S \in N$ is the non-terminal staring symbol.

Starting from the start symbol, the production of the grammar are derived, replacing any non-terminal symbol with one of the possible derivation on the right side.
A context-free grammar is equivalent to a non-deterministic push-down automaton, that is a finite automaton with a stack memory.
In the 2.1 is shown how a context-free grammar for the previous example can be written. The bold characters represent the non-terminal symbol of the grammar.
The syntax tree of the example can be constructed deriving the rules of the grammar and

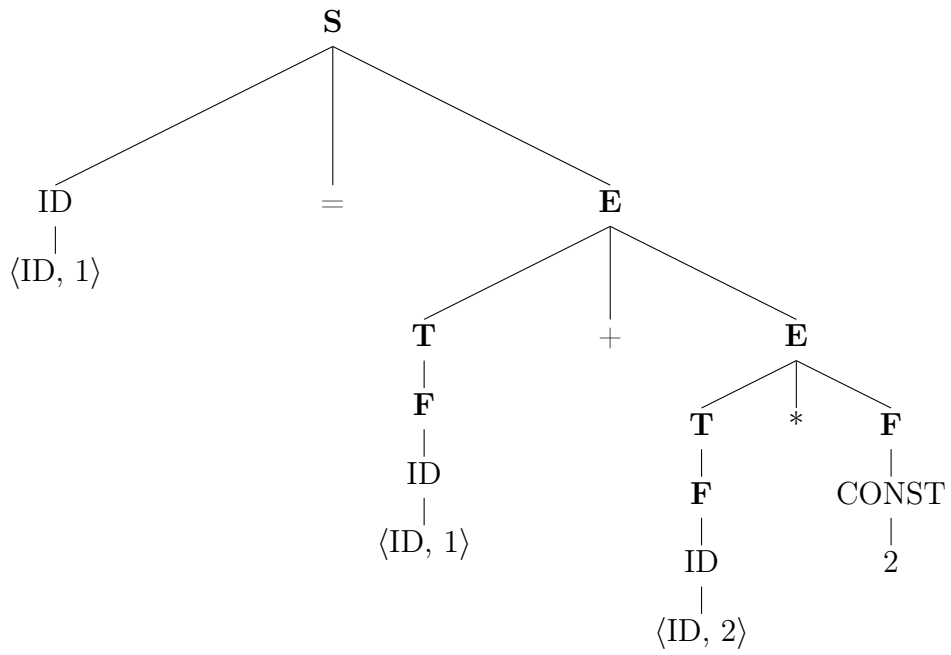


Figure 2.3: Concrete syntax tree derived using the context-free grammar 2.1

can be represented as the one in figure 2.3

$$
\begin{aligned}
&\mathbf{S} \rightarrow ID = \mathbf{E} \\
&\mathbf{E} \rightarrow \mathbf{E} + \mathbf{T} \mid \mathbf{T} \\
&\mathbf{T} \rightarrow \mathbf{T} * \mathbf{F} \mid \mathbf{F} \\
&\mathbf{F} \rightarrow (\mathbf{E}) \mid ID \mid CONST
\end{aligned}
\tag{2.1}
$$

Semantic analysis After the parser has build the syntax tree, the semantic analyzer checks that the source program is semantically correct, i.e. that the program respect the rules of the language. During the semantic analysis *type checking* is performed to ensure that each operator has a matching operand type. For example in C is not possible to index an array with a floating point number [32].

2.1.3. Intermediate representation

After all the semantic checks are successfully completed, the code is technically ready to be translated to machine code. Despite this, in order to firstly optimize the code and to keep the compiler modular, the code is first translated in an intermediate representation (IR).

Keeping the frontend and backend of the compiler separate, allows to have only one frontend module for each high-level language and one backend module for each target machine, improving the portability of the compiler. Suppose that the compiler backend is already available for a certain target machine M. If a new high-level language L needs to be compiled on M, with this approach, only the front end module of L has to be developed. Once L is translated to the IR, common for every language, the machine code for M can be generated without further changes.
Various different forms of IR exist, such as IR-Trees [14] or Three-Address Code [10], but the common required feature is that the IR needs to be easy to be produced from the syntax tree during the semantic analysis, easy to be translated to machine code, and express only simple operations to facilitate the implementation of the optimizations, but remaining at a sufficiently high level so that it is totally independent of the target machine and to not lose important information that can improve the quality of subsequent optimizations [32].

2.1.4. Code generation

After the intermediate representation of the code has been generated and the analyses, some of which are described in the following sections, to produce an optimized version of the IR, have been performed, the compiler can now enter the last phase of the compilation, also known as the *backend* of the compiler. The steps performed during this final phase, are no more independent of the target machine the compiler is producing the code for.

Target-dependent optimization Now that the compiler is aware of which architecture the program will run on, it can apply to the IR, specific target-dependent optimizations that cannot be generalized for other architectures. These last optimizations together with the ones performed in the optimization phase of the compiler are essential in order to produce a high quality result that can be executed efficiently.

Instruction selection The compiler have to convert the IR instructions in actual instructions that can be executed, mapping the IR instructions to the ones available for the correspondent target. Producing an acceptable result in terms of size and speed can be a complex problem that depends on the IR, on the target architecture and on the quality of code desired [10]. For example is easier to generate code starting from a lower level IR for an architecture that support more instructions and data-types.

Register allocation When the compiler, during the first phase, generates the IR code, assumes to have an infinite number of registers to store all the intermediate values produced by the instructions. In this way it is much simpler to work with the IR in order

to perform analyses or apply optimizations. But in reality an architecture with infinite register does not exist, in opposition with the majority of the architectures that have only few registers available. The compiler must then decide, for each point in the program, which values are kept in registers and which in memory. This problem can be divided in two sub-problems:

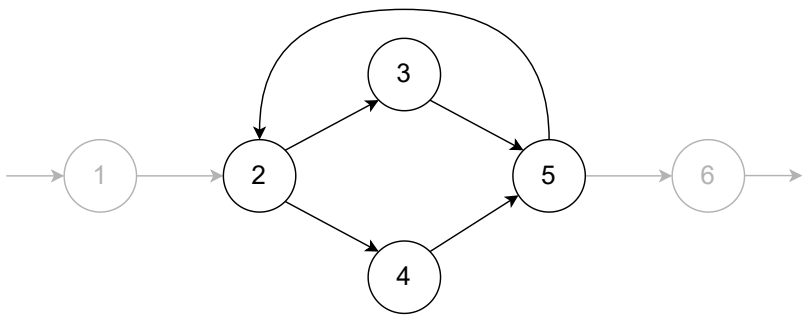
- **Register allocation** in which the compiler decides the set of variables that will be maintained in the register for each point in the program.
- **Register assignment** in which the compiler selects a register for each variable.

This problem is NP-Complete and it is often solved using heuristics, such as the graph coloring algorithm [10].

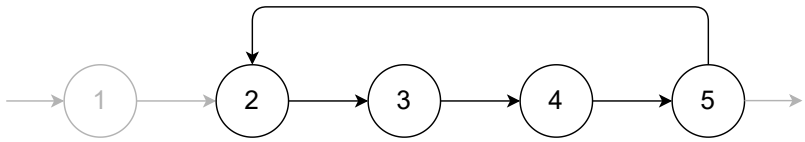
2.2. Control flow graph

Discovered by Frances E. Allen [12], control flow graphs (CFG) are a convenient way to represent the control flow of a program. In the LLVM framework the CFG is a fundamental structure used to maintain the intermediate representation of the program during the optimization phases. The use of a CFG helps to perform many analyses that rearguards the control flow or the dominance of the instructions.
Formally a CFG is defined as a directed graph $\mathcal{G}(\mathcal{B}, \mathcal{E})$ such that:

- There is a node $b \in \mathcal{B}$ for each instruction or *basic block* of the intermediate representation of the program.
- There is an edge $(b_i, b_j) \in \mathcal{E}$ if the statement $stat_i$ is executed immediately before the statement $stat_j$
- If there is an edge $(b_i, b_j) \in \mathcal{E}$ the node b_j is called an *immediate successor* of b_i, and b_i is the *immediate predecessor* of b_j
- The function $f_s(b_i) = \{b_j \mid (b_i, b_j) \in \mathcal{E}\}$ returns the immediate successors of the node b_i
- The function $f_p(b_i) = f_s^{-1}(b_i) = \{b_j \mid (b_j, b_i) \in \mathcal{E}\}$ returns the immediate predecessors of b_i
- There are two additional nodes b_{in}, the entry node, which has no predecessors and b_{out}, the terminal or exit node, which has no successors.



(a) The set of nodes {2,3,4,5} is the maximal SCC



(b) A CFG with a circuit composed by the nodes {2, 3, 4, 5}

Figure 2.4: Two examples of CFG in which is possible to observe to different properties of them

Basic block A basic block is a sequence of instructions that doesn't contain any type of branch, i.e. the instructions in basic block are always executed one after the other. At the end of a basic block there is always a branch.
In CFG a basic block can be recognized as a sequence of nodes $\langle b_0, ..., b_n \rangle$ such that the only existing edges are $(b_0, b_1), ..., (b_{i-1}, b_i)$ except for the edges entering the first node of the basic block and the edges exiting the last node.

Connected graph A CFG is connected if any node in the graph can be reached by any other node by successive application of f_s and f_p.

Strongly connected component/region A strongly connected component (SCC) is a subset of the nodes $\in \mathcal{B}$ in which any node of the SCC can be reached by any other node of the same SCC by successive application of f_s, so it exists a directed path from any node to any other node of the SCC (Figure 2.4a).
An SCC is maximal if there are no other nodes in $\mathcal{G}$ that can be added to the SCC without losing the strongly connected property.

Circuit A circuit is a special case of an SCC with a strict ordering and represent a loop in the CFG. A circuit is a path in which $b_0 = b_n$ (Figure 2.4b). A circuit is simple if the nodes in the path are not repeated.

2.2.1. Dominance relation

An important property that can be derived from a CFG is the dominance relation between the nodes of the CFG. This property can be used to discover and optimize the loops in the code, and to build the static single assignment form (Section 2.3) of the intermediate representation.

Definition 2.2.1 (Dominator)**.** A node b_i *dominates* a node b_j if every path of directed edges from b_{in} to b_j must go through b_i. Every node dominates itself. A node b_i *strictly dominates* b_j if b_i dominates b_j and $b_i \neq b_j$ [14].

Definition 2.2.2 (Immediate dominator)**.** A node b_i is the *immediate dominator* of b_j (with $b_i \neq b_j$) if b_i dominates b_j and $\forall\ b_n$ such that b_n dominates $b_j \Rightarrow\ b_n$ dominates b_i. Every node has exactly one immediate dominator, except the entry node b_{in} that has no dominators [12, 14].

Dominator tree

A dominator tree is a tree-like structure constructed starting from a CFG, and with all its nodes. The edges in the dominator tree represent the **immediate** dominance relation between two nodes. In the dominator tree each node has only one incoming arch, since a node can have only one immediate dominator. A predecessor in the dominator tree does not always correspond to the predecessor of that node in the CFG [14]. An example of CFG and its corresponding dominator tree is presented in figure 2.5

Dominance frontier

The dominance frontier of a node b_i is the set of nodes **b** such that b_i dominates a predecessor of **b** but does not strictly dominate **b** [14]. In other words the dominance frontier of b_i are the nodes outside the area dominated by b_i, but that are reachable from the nodes in that area.
A simple and efficient algorithm for computing the dominance frontier of a node exist and the pseudocode from [14] is reported in 2.1. The function idom(b) return the immediate dominator of b, and the function succ(b) return all the successor of b in CFG.
An example, using the algorithm 2.1, of the computation of the dominance frontier of the node 5 of the CFG represented in figure 2.6, is:

1. In the first for-each loop the nodes that are successors of 5 are {6, 7, 8, 4, 12, 13}. For the nodes {6, 7, 8} the immediate dominator is the node 5, so they are **not** included in S. {4, 12, 13} have the node 1 as immediate dominator, so they are

added in S.

2. In the second loop, the children of the node 5 in dominator tree are {6, 7, 8}. As we can see, the dominance frontier for the node 6 is composed by all its successors {5, 7, 8, 4, 13, 12} as 6 does not dominate any node except itself.

3. Of these nodes, the node 5 **dominates** {5, 7, 8}, but 5 is also the considered node (b_n) itself so the nodes that are finally included in S during the nested loop are {4, 13, 12}, that was already present after the first step.

4. The steps 2 and 3 for the other two nodes (7, 8) is similar to the one of node 6.

5. Finally, the computed dominance frontier for the node 5 is composed by the nodes {4, 12, 13} that are marked in figure 2.6

Algorithm 2.1 Dominance frontier computation

```
1:  S ← {}
2:  for each node b_i in succ(b_n) do
3:      if idom(b_i) ≠ b_n then
4:          S ← S ∪ {b_i}
5:      end if
6:  end for
7:  for each child c of b_i in the dominator tree do
8:      DF_c ← dominance frontier of c
9:      for each w ∈ DF_c do
10:         if b_n does not dominate w or b_n = w then
11:             S ← S ∪ {w}
12:         end if
13:     end for
14: end for
15: return S {At this point S contains the dominance frontier of b_n}
```

2.3. Static single assignment

The static single assignment form, or SSA form, is a way of writing the intermediate representation where each variable has only one definition. The advantages of using the SSA form comes from the fact that many data flow analyses and optimizations requires that the compiler maintain a data structure to represent the definitions and the uses for each variable, and using the SSA form can substantially reduce the time and space

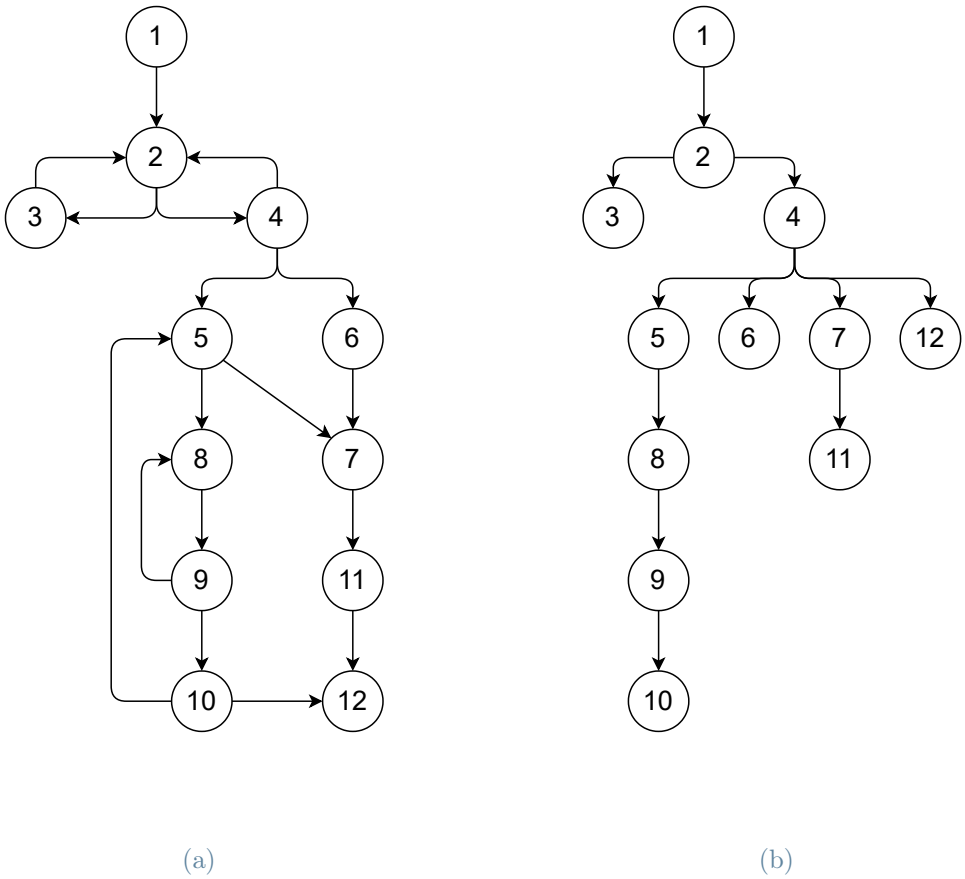


Figure 2.5: In the figure 2.5a there is a CFG and in the figure 2.5b there is its correspondent dominator tree.

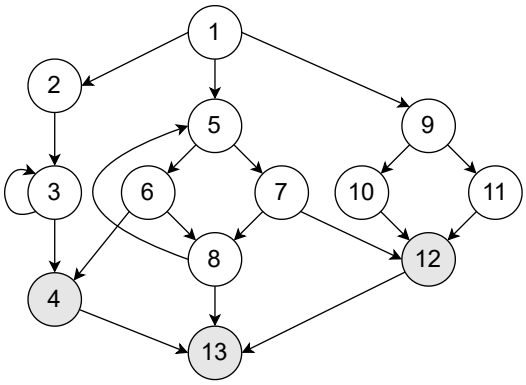


Figure 2.6: A control flow graph from [14]. In gray the nodes of the dominance frontier of the node 5.

complexity needed to maintain that data structure, from quadratic to linear [14, 18]. Furthermore, the use of SSA form simplifies the algorithms that perform such analyses or optimizations, because it can be assumed that for each use of a variable there is only one definition.

2.3.1. Construction of the SSA form

If we consider only a single basic block (2.2) the conversion from a non SSA IR to an SSA IR is quite simple. As shown in figure 2.7 the only task to perform is to rename the variables assigning to each of them a new unique name at every definition, and change the subsequent use of that variable with the most recent definition.
The problem arises when we consider more than one basic block, and particularly when a basic block has more than one entering edge in the CFG, and more than one predecessor define the same variable. In this case more versions of the same variable is entering the node, and the concept of "most recent definition" loses its meaning.
To solve this problem, we introduce the special notation of $\phi - function$ [14], a function that is inserted at the beginning of each basic block that present such a problem. The $\phi - function$ takes as parameters all the possible definitions of a variable coming from different paths, and return the value of the correct definition based on which CFG path the program took during the execution.
The steps for converting an IR to the SSA form are:

1. Insert the $\phi - functions$ where needed.
2. Rename all the variables.

$\phi - function$ insertion

A naif approach to insert the $\phi - functions$ is to add a new $\phi - function$ for each variable, every time there is a join point in the CFG, but if a variable reaches a node from different paths but with the same definition, i.e. the variable is not redefined in any of these paths, there is no need to insert a $\phi - function$ for that variable.
Two others, and more clever, algorithms to insert the $\phi - functions$ exist. One of them is the **path-convergence criterion** and the second one more efficient algorithm is the **dominance frontier criterion** based on the definition of dominance frontier 2.2.1 [14].
The dominance frontier criterion algorithm is briefly described in these steps:

1. For each variable v defined in a node b insert a new $\phi - function$ on the nodes of the dominance frontier of b ($DF(b)$)

$$
\begin{array}{ll}
x \leftarrow y + 3 & x_0 \leftarrow y_0 + 3 \\
y \leftarrow 4 * x & y_1 \leftarrow 4 * x_0 \\
x \leftarrow x + 2 & x_1 \leftarrow x_0 + 2 \\
z \leftarrow x + y & z_0 \leftarrow x_1 + y_1 \\
x \leftarrow z & x_2 \leftarrow z_0 \\
\text{(a)} & \text{(b)}
\end{array}
$$

Figure 2.7: On the left there is a pseudocode with some assignment, and on the right the same pseudocode in SSA form

2. Since a $\phi - function$ is a new definition itself, repeat the first step for each node of $DF(b)$
3. The process terminate when all the definition of v has been considered.

2.4. Data Flow Analysis

Data flow analysis (DFA) is a set of different static analyses all aimed at understanding how the data flows between the instructions during the execution of a program, and provide essential information for the application of many code optimizations [10]. Furthermore, DFA is fundamental to build tools that protect ciphers implementations, such as Cryptoshiled described in section 1.5, as the analyses that identifies the vulnerable instructions are all DFA.

Data flow analysis is conservative Since DFA is a static analysis, i.e. an analysis executed "observing" the code without executing it, the results of the analyses will be just an approximation of the reality. DFA is *conservative*, as the results of the analyses **always** preserve the correct behavior of the program, even though they can be imprecise [10]. For example an analysis might consider all the paths in a program, producing possible additional errors, even if some of those path are unreachable during the execution. In this way the analysis may have optimized the program less, but ensuring the correctness of the results.

Definitions and Uses The control flow graph can be enriched with the sets of the variables that are defined and used inside each node. For a node b the two sets can be defined as **def**(b) containing all the variables for which there is at least an assignment inside the node, and **use**(b) containing all the variables that are used at least in one instruction inside the node. An example of CFG with definitions and uses is shown in figure 2.8

2.4.1. Data Flow Equations

The goal of the DFA is to find the status of the interested data-flow values, before and after each node of the control flow graph. For a node b, the data-flow values entering b are defined as **IN[b]**, while the ones exiting b are defined as **OUT[b]**. In order to find the solutions for each node of the CFG, two types of constraints must be solved. These constraints form the so called *data-flow equations* [10].

Transfer function The transfer function f_s describes the relationships between the data-flow values before and after a node, i.e. how the information is transformed inside a single basic block of the program.
The analysis can be distinguished in:

- **Forward analysis** that is when the information propagates in the same direction of the execution paths, i.e. from the start of the program towards the end.

$$OUT[b] = f_s(IN[b])$$

- **Backward analysis** that is when the information propagates up the execution paths, i.e. form the end of the program towards the start.

$$IN[b] = f_s(OUT[b])$$

Control-flow constraint A control-flow constraint describes how the information propagates between the nodes of the CFG.
For the **forward analysis** the information entering a node b ($IN[b]$), depends on the information exiting its predecessors ($pred(b)$), so the control-flow constraint is

$$IN[b] = \bigcup_{\forall p \in pred(b)} OUT[p]$$

Instead for the **backward analysis** the information exiting a node b ($OUT[b]$), depends on the information entering its successors ($succ(b)$), so the control-flow constraint is

$$OUT[b] = \bigcup_{\forall s \in succ(b)} IN[s]$$

2.4.2. Reaching Definitions

Reaching definitions is a *forward* DFA which has as its purpose to determine, for each node, what are the definitions of variables that can reach that node without being over-

written.
More formally, a definition v_b of a variable v in a node b *reaches* the entrance of a node q if there is a path from b to q such that v_b is not *killed* along that path [10], i.e. that the path does not traverse any other node $k \neq b$ where v is defined.
If a node b defines a variable v, producing the definition v_b, we say the node b *generates* the definition v_b and *kills* any other definition of v_q with $b \neq q$.

$$\begin{cases} gen(b) = \{v_b\} \\ kill(b) = \{v_q \mid v \in def(b) \ \wedge \ v \in def(q) \ \wedge \ b \neq q\} \end{cases} \tag{2.2}$$

The **data flow equations** for the reaching definitions of a node b are the 2.3, where the function *kill* is defined as 2.2

$$\begin{cases} OUT[b] = gen(b) \cup (IN[b] \setminus kill(b)) \\ IN[b] = \bigcup\limits_{\forall p \ \in \ pred(b)} OUT[p] \end{cases} \tag{2.3}$$

As reaching definitions is a forward analysis, the solution to the data flow equations can be obtained iteratively starting from the first node of the CFG initialized as

$$IN[b_{in}] = \varnothing$$

until a fixed point is found, that is when an iteration does not change any set. Reaching definitions can be used to delete unused variables, or for detecting if a variable is used before its definition [10].

2.4.3. Liveness Analysis

Another fundamental DFA is the liveness analysis, a *backward* analysis aimed to understand for each point in a program if a variable is *live*, i.e. its value will be used again before any new definition in at least one path of the CFG, or *dead*, i.e. its value will not be used anymore.

Definition 2.4.1 (Live variable)**.** A variable v is *live* in input or output of a node b_i if in the CFG there is a path from b_i to b_j such that b_j use v ($v \in use(b_j)$) and the path does **not** traverse any other node $b_k \neq b_i$ that defines v ($v \in def(b_k)$) [10].

The data flow equations to solve the liveness analysis for a node b are the 2.4, in which the sets $IN[b]$ and $OUT[b]$ are respectively renamed $live_{in}(b)$ and $live_{out}(b)$ to better clarify their meanings.

$$\begin{cases} live_{in}(b) = use(b) \cup (live_{out}(b) \setminus def(b)) \\ live_{out}(b) = \bigcup_{\forall p \in succ(b)} live_{in}(p) \end{cases} \tag{2.4}$$

As for the reaching definitions, the solutions of the data flow equations can be found applying the rules iteratively until a fixed point is reached. Instead, contrary to the reaching definitions the analysis starts from the final node b_{out} which is initialized as

$$live_{out}(b_{out}) = \varnothing$$

which means that no definition exit the program. Examples of application of liveness analysis are the simplification of the register allocation, because if two variables are not simultaneously live, they can share the same register, or to find useless defines of variables that will never be used, and so they can be deleted.

2.5. LLVM

LLVM is a compiler framework and an umbrella project for a large collection of different subprojects, like the Clang C/C++/Objective-C compiler or the very fast and efficient LLD linker, and it is used both in academic research and in production. The development of LLVM began in 2000 as a research project at the University of Illinois but from than it became very popular. The success of LLVM, as stated by the original project author Chris Lattner in [3, 28], is mainly to be attributed to the design of its architecture.
LLVM, still today, is one of the few or even the only one piece of software that allows to build a compiler in a totally modular way, in fact LLVM is developed following the tree-phase design (Figure 2.1) in order to be flexible and reusable.

LLVM is organized as set libraries that expose a wide range of different APIs. Moreover, each phase of LLVM is a pipeline composed by different passes built to be independent of each other. A developer can then build a personalized pipeline with the passes already available or develop new passes which integrates with the existing ones without having to modify other parts of the tool-chain. The compiler or tool that is developed using LLVM will be linked only with the libraries of the code that is actually used.

Internally LLVM uses different types of data structures and intermediate representations. The LLVM Core libraries use a control flow graph (Section 2.2), the LLVM IR, and the LLVM MIR (Machine Intermediate Representation) in order to perform analyses, transformations and code generation. Other representation can be used such as the abstract

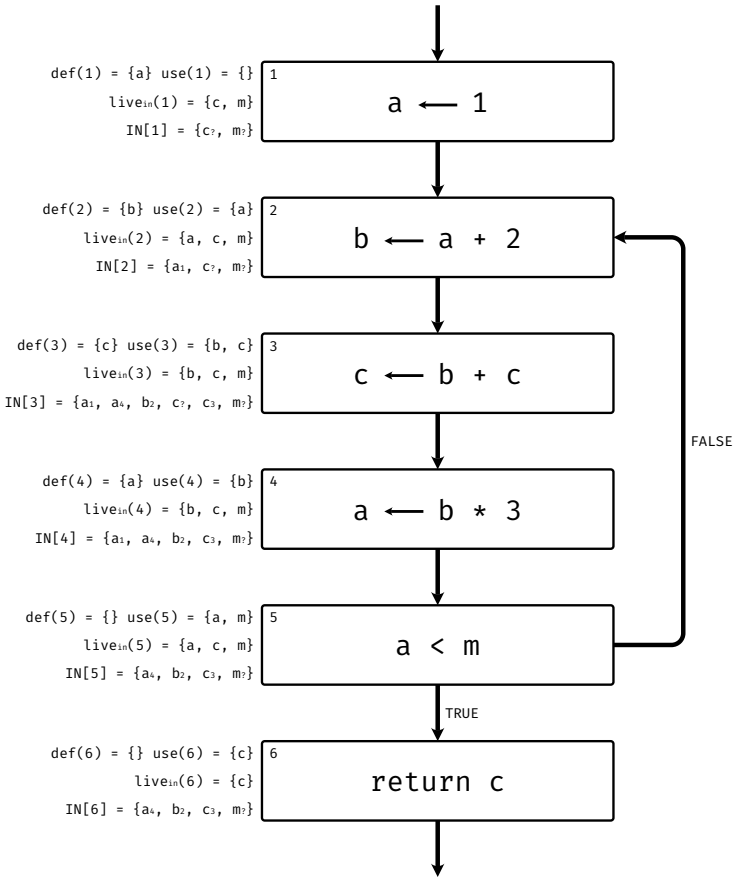


Figure 2.8: Example of data flow analysis showing the control flow graph and for each node the definitions the uses and the results of the liveness analysis and the reaching definitions. The variables c and m are parameters that are passed to this function from external.

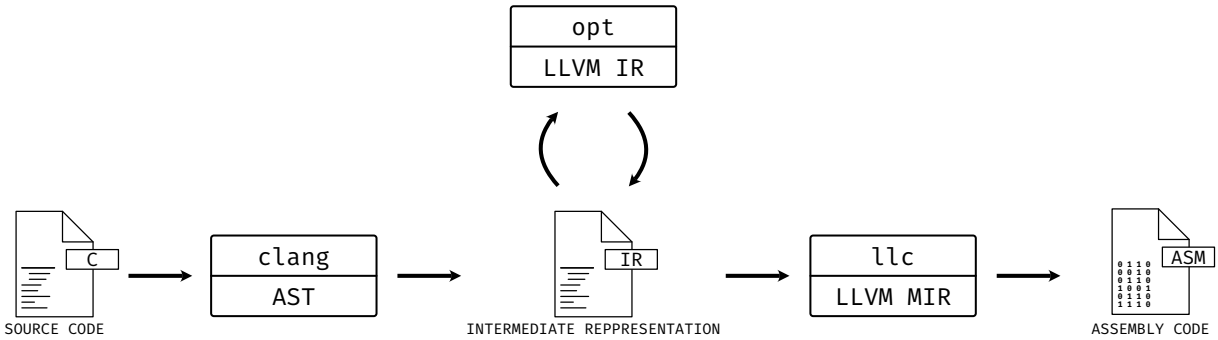


Figure 2.9: Simplified scheme of the stages and the IRs used during a compilation of a C file using Clang

syntax tree (AST), i.e. a condensed version of the syntax tree presented in section 2.1.2, used by Clang before generating LLVM IR. A simplified scheme of typical compilation pipeline using Clang is shown in figure 2.9. *opt* is the source language and target independent llvm optimizer, while *llc* is the llvm static compiler, which translate LLVM IR in LLVM MIR, applies the target-dependent optimizations and produce an assembly file for a specific architecture. The assembly file can then be passed through a native assembler, such as *llvm-mc*, and a linker, such as the llvm linker *lld*, to generate a native executable [1].

2.5.1. LLVM IR

LLVM IR is one of the most important aspect of the design of LLVM. It's used mainly during the middle optimization phase. It's a first class language, with a low level RISC-like instruction set. The instructions are in three address form which means that each instruction takes some inputs and produce a result, each of one is stored in a different register, which are of infinite number [3].
LLVM IR is **always** in SSA form (Section 2.3) without any exception. A fragment of code that doesn't respect SSA form is invalid, in fact trying to reassign a value to temporary variable (register) will produce an error.

The tree forms of LLVM IR

Tree isomorphic forms of LLVM IR exists: [3]

- **In-Memory** This form is used internally by LLVM during the compilation phases. The LLVM IR is maintained in a control flow graph, and analyses or transformations can be done easily using the APIs provided by LLVM.

- **Bitcode** The bitcode (.bc) is a dense binary file format (similar to the more famous Java bytecode) that store the IR on disk. This is the most efficient way to store LLVM IR.
- **Textual** The textual format (.ll) is another way to store LLVM IR on disk, but in human-readable form that can be used to debug or even manually modify the code produced.

LLVM provides different tools to convert a form to another and vice versa. The *llvm-as* is used to convert the textual form to the bitcode form, and the *llvm-dis* to do the contrary.

Type system

An important aspect of LLVM IR is that the language is *strongly typed*, which means that anything has a type and conversions from one type to another requires specific instructions even with similar types such as different sized integers. This design choice was made because having a typed IR enables many analyses and transformations that cannot be performed without having to do extra analyses [1].

The type system is organized in types that can be returned as a result of an instruction, called *first class types*, which can be *single value* types, i.e. the ones that can be stored directly in a register, and *aggregates* types such as arrays or structures. Other types that cannot be returned by an instruction are the `void` and `function` types.

An example of single value type is the integer type, expressed with the syntax `iN` where N is the bit width, usually used in the form `i32` or `i64`, but it can range from 1, used to express boolean values, to 2^{23}.

Pointer type In LLVM pointers have the *opaque* `ptr` type, which means that is not directly possible to know the type of the value pointed by a specific pointer, also known as *pointee type*. In older version of LLVM (prior to LLVM 15) pointers had a type, but for various different problems related to some optimizations, the adoption of LLVM from other source languages other than C, and to simplify the IR removing useless bitcast instructions, it has been seen that the costs of keeping the pointee types outweigh the benefits and so they were removed [2].

Layout

LLVM programs are composed of *modules*, where each module is a unit of compilation (Each .ll or .bc file contains one module), that can be linked together. Each module contains information about the target and global symbols that are global variables and

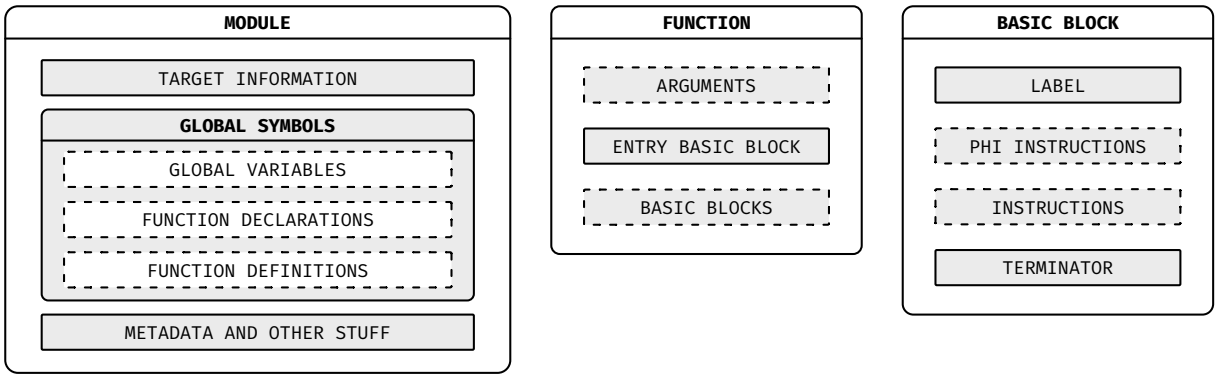


Figure 2.10: A simplified diagram of the LLVM IR layout.

function declarations and definitions. Functions can have zero or more arguments, and if the function is a definition it must have at least one basic block, called *entry block*. Each basic block must start with an identification label e end with a terminator instruction, that can be a branch instruction to another basic block or a return instruction. Furthermore, inside a basic block, possible phi-instructions must be placed before any other instruction. A diagram of this layout is presented in figure 2.10.

Instruction Set

The instruction set of LLVM IR is organized in different classes of instructions. An instruction contained in a reachable basic block must be *well-formed*, i.e. that the instruction must respect the SSA form where it dominates all its uses. For example the instruction |`%x = add i32 1, %x`| is not well-formed as the definition of `%x` doesn't dominate all its uses.

Here is reported a subset of the most used instructions:

Terminator instructions These are the instructions used at the end of a basic block. As I'm writing there are 11 different instructions in this class, but the two more used are the

- *Branch instruction*: used to transfer the control flow to different basic block. The instruction can branch conditionally or unconditionally.
- *Return instruction*: used to return after a function call, and optionally accepts as argument the value that has to be returned.

```
br i1 %cond, label %iftrue, label %iffalse   ; Conditional branch
br label %dest                               ; Unconditional branch

ret i32 %x                                   ; Return instruction
```

Unary operations The only instruction in this class is the `fneg` instruction that takes as input a single or vector of floating point numbers and return the same number with the sign flipped.

Binary operations are all the instructions that computes arithmetic operations. Accepts two values of the same type and return a value of the same type. The types accepted can also be vectors. The following are all the instructions in this class. All the instructions exists in integer form or in floating point form (with the starting f). The division and the reminder exist also in unsigned or signed *integer* form.

- `[f]add`: returns the sum of the arguments
- `[f]sub`: returns the difference of the arguments
- `[f]mul`: returns the product of the arguments
- `<f|u|s>div`: returns the quotient of the arguments
- `<f|u|s>rem`: returns the remainder from the division of the arguments

```
%sum = add i32 5, %var          ; %sum = 5 + %var
%result = mul i32 %sum, %x      ; %result = %sum * %x
```

Bitwise binary operations As for the binary operations, these instructions accept two operands of the same type and return a value of the same type of the operands. All the instructions in this class accept integers or vector of integers.

- `shl`: returns the first operand shifted to the left a specified number of bits.
- `lshr`: logical shift right. Returns the first operand shifted right a specified number of bits and fill with zeros.
- `ashr`: arithmetic shift right. Returns the first operand shifted right a specified number of bits with sign extension.
- `and`, `or`, `xor`: return the bitwise logical *and* / *or* /*xor* between its operands.

Memory operations This is the class of instructions used to read, write, allocate memory, and compute the memory addresses. The most important ones are:

- `alloca`: used to allocate memory on the stack of the function currently under execution. Can be used to allocate any type of data structure, and return the pointer to the allocated memory.
- `load`: read from specified address in memory and return the value.

- `store`: writes the first operand to the address of memory specified by the second operand. The store instruction does not return any value.
- `getelementptr`: computes the address of an element in an aggregate data structure. It's used to access the content of arrays, matrices and other more complex structures. The instruction does **not** return the content of the memory but only computes the address that can be used with a `load` instruction to obtain the desired value.

```
%struct.point = type { i32, i32 }

%p = alloca %struct.point, align 4  ; Allocate the struct
; Compute the address of the first value
%x = getelementptr inbounds %struct.point, ptr %p, i32 0, i32 0
store i32 4, ptr %x, align 4        ; Store 4 in the first value
; Compute the address of the second value
%y = getelementptr inbounds %struct.point, ptr %p, i32 0, i32 1
store i32 5, ptr %y, align 4        ; Store 5 in the second value
```

Conversion operations These are the instructions used to convert types, as LLVM IR is a strongly typed language.

- `trunc .. to`: Truncates an operand to a specified type. The first operand type width must be greater of the width of the specified destination type.
- `zext .. to`: Zero extend the operand to the specified type.
- `sext .. to`: Sign extend the operand to the specified type.

```
%x = trunc i32 257 to i8 ; returns i8:1
%y = zext i32 42 to i64  ; returns i64:42
%z = sext i8 -1 to i16   ; returns i16:65535
```

Other operations Include instructions to compare values, call a function and the phi-function instructions.

- `<i|f>cmp`: returns a boolean value based on the comparing of its operands.
- `call`: call the function specified with optional parameters.
- `phi`: used at the beginning of a basic block to implement the ϕ-function used in the SSA form (See section 2.3). Here below there is the syntax of this instruction. It returns the value correspondent to label of the basic block that had defined the value.

```
<result> = phi <ty> [<val0>, <label0>], [<val1>, <label1>], ...
```

2.5.2. LLVM APIs

The LLVM framework is developed using C++ with extensive use of the STL (Standard Template Library) and with the use of a custom RTTI (Runtime Type Identification). For this reason to cast or to test the LLVM Types custom functions exist:

- `isa<>`: used to test if a reference or a pointer points to an instance of the specified type.
- `cast<>`: used to cast a type to the specified one, producing an error in case the passed type is not an instance of the right type.
- `dyn_cast<>`: used to test and cast a type to the same type. In case the cast is successful it returns a pointer to the instance, otherwise it returns `nullptr`

Listing 2.1: Example of custom LLVM RTTI

```
if (isa<LoadInst>(I)) {
    // Here I know that *I is a load instruction
}

// This statment may produce an error if *I is not an instance of ↩
    LoadInst
Value* OP = (cast<LoadInst>(I))->getPointerOperand();

if (LoadInst* LI = dyn_cast<LoadInst>(I)) {
    // Do something with LI
}
```

The most important classes in to work with LLVM APIs are:

- **Value**: This is the most important class of LLVM. Many other classes is derived from this class, such as Instructions, Functions, and Constants.
- **Module**: The Module class is the container for functions and global values. With this class is possible to remove or create new global variables, search a specific function by its name or iterate over all the existing functions using `Module::begin()` and `Module::end()`
- **Function**: This class represent a single function in the IR with some possible arguments, and, similarly to the Module class, can act as an iterator to iterate over the BasicBlocks and the Arguments using respectively `Function::begin()` and `Function::end()`, and `Function::arg_begin()` and `Function::arg_end()`
- **BasicBlock**: This class is a container for the IR instructions, and can be used to

iterate over the instructions using `BasicBlock::begin()` and `BasicBlock::end()`. BasicBlock provides also methods to explore the control flow graph accessing its successors and predecessors.

- **Instruction**: This is the superclass of all the instruction types.
- **User**: This is superclass for the Instruction class and exposes its operands. Usually it is used to access an instruction that *use* a certain value.

A much detailed view of some of the most important LLVM APIs classes hierarchy is shown in figure 2.11.

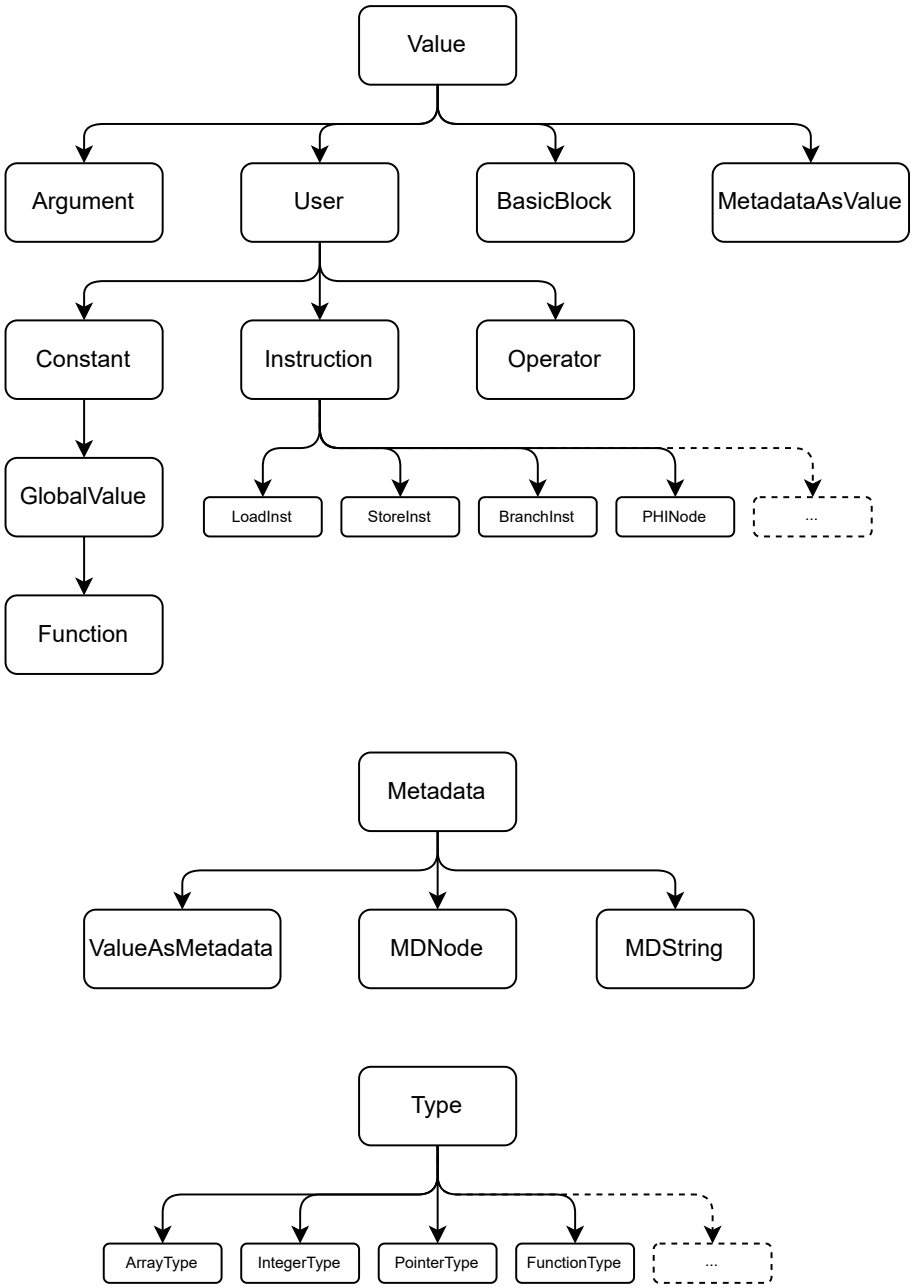


Figure 2.11: Hierarchy of some of the most important LLVM APIs classes

3 | Software architecture

This chapter contains the description of the work done to develop a new set of passes for the LLVM compiler framework (Section 2.5) which extends the capabilities of Cryptoshield, described in section 1.5, to provide a way to *dynamically* analyze the power consumption of a cipher algorithm in each point of the program.

The tool is developed to take as input a high-level source code (such as C) and to produce as output a binary file that can be used as a library to obtain different information for each instruction of the cipher algorithm, such as the intermediate value calculated, the time needed to compute the instruction from the beginning of the algorithm, and the *Hamming distance* between two intermediate values that has written in the same register consecutively. The data produced is enriched with other information about each instruction to help to understand the role that it has in the cipher algorithm.

The chapter is divided in two main sections. In the first section is described the process to *port* the Cryptoshield source code to a newer version of the LLVM framework, the reasons behind these decisions and how the code was adapted to work. The second section explain instead the modifications and the extension performed to develop the tool.

3.1. Cryptoshield porting

The newly developed tool exploit the analyses already developed inside Cryptoshield to obtain information such as the secret key and plaintext instruction dependencies or to know if an instruction is part of a round of the cipher.

Cryptoshield was developed using an old version of LLVM (version 3) and in the years the framework is evolved to improve performance and produce better code optimizations. This changes however come with a major drawback which is that some new features are incompatible with the code written to work with older versions. In fact, as the creator of LLVM, Chris Lattner, wrote in [3], one of the major aspect of LLVM is to remain nimble, i.e. that the developers of the framework can make, in any time, changes to the APIs without worrying about backward compatibility.

Furthermore, due to the fact that the tool was build on version of LLVM several years old, many libraries and dependencies are not supported anymore, so the compilation of the tool is difficult. For these reasons porting Cryptoshield to a recent version of LLVM was the right choice both to simplify the development of the new features presented in this thesis, and to keep the tool accessible in the future before other changes to the framework make a future porting too complicated. The new version of LLVM used to port and later extend the Cryptoshield tool is the 18.1.4.

Two of the major difficulties encountered during the porting of Cryptoshield were the impossibility to run the old version to compare the results of the analyses and the lack of precise documentation regarding the implementation, in particular of the changes made to clang to introduce the new annotations, such as the attributes to identify the parameters of the key and plaintext and the pragma to identify the rounds of the cipher.
Beside small changes, such as replacing some custom functions that no longer exist in LLVM with standard C++ functions existing in the version of the language used by the new version of the framework, or changing the control flow in some passes of clang, the biggest changes were made in the optimization passes and involved changing the LLVM pass manager and handling the new opaque pointers.

3.1.1. Pass Manager

As explained in section 2.5, each of the phases of LLVM is organized as a pipeline of different passes, that are managed by a *pass manager*. In 2019, with the version 9.0 of LLVM, the use of a new pass manager (new PM) was introduced as an experimental option for managing the passes in the optimization phase. In subsequent version of the framework, the new PM gradually replaced the old one (legacy PM), becoming the default pass manager in LLVM 12.0 and the only option with the release of LLVM 15.0.

An optimization pass can be of tree type that acts at different granularity:

- **Module pass**: Is the most general pass that operates on the entire compilation unit.
- **CGSCC pass**: The *CallGraphSCCPass* operates on the strongly connected components 2.2, and traverse the program bottom-up in the call graph.
- **Function pass**: Is executed for each function in a program.
- **Loop pass**: Is executed for each loop in a function.

The tasks of the PM is to maintain a pipeline of execution, manage the dependencies

between different passes, and caching the results of the analyses in order to speed up the compilation.
There are tree substantially improvements achieved with the new PM [21]:

1. The first one is that now analysis passes, i.e. passes that return a result produced analyzing the IR, and transformation passes, i.e. passes that apply an optimization, are now distinguished and the results of the analyses are managed by an analysis manager that can cache arbitrary analyses, even of a single function, reducing useless cache invalidation.

2. The second major improvement is that now both the execution pipeline and the dependencies' declaration can be done dynamically, reducing the boilerplate code, and allowing to create external plugin for LLVM.

3. The new PM is now able to parallelize independent passes.

The examples 3.1 and 3.2, of fragments of code taken from the real implementations of Cryptoshield, show how an existing pass can be converted from using the legacy PM to the new PM. The main differences are that the super class of the pass is changed from the generic `FunctionPass` to `AnalysisInfoMixin<>` that indicates that this is an analysis pass (For transformation passes the super class to use is `PassInfoMixin<>`), the function `getAnalysisUsage`, used to declare at compilation time the dependencies between passes, no longer exists, and the required results from other analyses are now managed by the `FunctionAnalysisManager`. Furthermore, also the global function `X` to statically register the pass is no longer present as now passes are dynamically registered in the pipeline.

Listing 3.1: Function analysis pass using the legacy PM

```
class AttackCandidatesPass : public FunctionPass {
public:
  static char ID;

  AttackCandidatesPass() : FunctionPass(ID) {}

  bool runOnFunction(Function &F) override;
  void getAnalysisUsage(AnalysisUsage &AU) const override;
};

void AttackCandidatesPass::getAnalysisUsage(AnalysisUsage &AU) const {
  AU.setPreservesAll();
  AU.setPreservesCFG();
  AU.addRequired<InputDependenciesPass>();
  AU.addRequired<CtxDependencyDepthPass>();
```

```
16 }
17
18 bool AttackCandidatesPass::runOnFunction(Function &F) {
19   const InputDependenciesResult &Id =
20       getAnalysis<InputDependenciesPass>().getAnalysisResult();
21   const CtxDependencyDepthResult &Cdd =
22       getAnalysis<CtxDependencyDepthPass>().getAnalysisResult();
23
24   // Code of the analysis
25
26   return false; // The pass did not change the code
27 }
28
29 char AttackCandidatesPass::ID = 0;
30
31 static RegisterPass<AttackCandidatesPass> X(
32   "attack-candidates",
33   "Identify points where an attack could be mounted",
34   true, // Only looks at the CFG
35   true  // Is an analysis
36 );
```

Listing 3.2: Function analysis pass using the new PM

```
1 class AttackCandidatesPass : public AnalysisInfoMixin<↩
    AttackCandidatesPass> {
2
3 public:
4     AttackCandidatesResult run(Function &F,
5                                FunctionAnalysisManager &AM);
6
7     static llvm::AnalysisKey Key;
8 };
9
10 AnalysisKey AttackCandidatesPass::Key;
11
12 AttackCandidatesResult AttackCandidatesPass::run(Function &F,
13                                                  FunctionAnalysisManager &AM) {
14
15     const auto &Id = AM.getResult<InputDependenciesPass>(F);
16     const auto &Cdd = AM.getResult<CtxDependencyDepthPass>(F);
17
18     // Code of the analysis
19 }
```

The source code of the LLVM framework is very large, and compiling it requires a lot of resources and time. To partially solve this problem, all the optimization passes of Cryptoshield have been ported as an out of source plugin, which means that the tool is compiled as a dynamic library that is loaded during the execution. Unfortunately there is no way to include in the dynamic library also the changes needed for adding and managing the new annotations (the attributes to identify the secret key and the plaintext, and the pragma to identify the cipher rounds), so these changes were made by directly modifying the LLVM and Clang source code.

3.1.2. Opaque Pointers

As explained in section 2.5.1, in older version of the LLVM framework, pointers had a type, such as `i32*`, but in recent versions it has been replaced by a generic `ptr` type.
This "small" change caused some problems, as this information is used in many analyses.

The solution to this problem comes from the fact that pointers are used to access the memory in a particular location, but the instructions that read or write into the memory, respectively the `load` and the `store` instructions, need to know the size of the memory to read or write. At the same time that pointers types became opaque, the instructions `load` and `store` gained the type to access the memory correctly.
For example a `load` instruction that read an integer (`i32`) from the location of the pointer `%var`, in a version of LLVM without opaque pointers is written

```
%0 = load i32* %var, align 4
```

while the same instruction in a recent version of LLVM is written

```
%0 = load i32, ptr %var, align 4
```

It was therefore possible to develop an algorithm that, taken a pointer value as input, explore all of its uses to find a `load` or `store` instruction and so returns the type of the pointed value. In 3.1 there is the pseudocode of this algorithm that for each *user* `U`, i.e. the value that *uses* another one as operand, of `Ptr` (the pointer which we interested to know the type) distinguish four different cases:

- **U is a GEP instruction**: In this case either the GEP is calculating the address of a struct, and in this case the type is already available, otherwise we call the algorithm recursively with U.
- **U is a load instruction**: In this case if the type of the load is itself a pointer we are against a pointer of pointer and we search recursively using U. Otherwise the type of the load instruction is returned.

Algorithm 3.1 Algorithm to find the type of a pointer

```
findPointerType(Ptr: Value*):
for each user U of Ptr do
  if U is a GetElementPtrInst then
    if the source element is a struct then
      return the type of the source element
    end if
    return findPointerType(U)
  end if

  if U is a LoadInst then
    if the type of U is a PointerType then
      return findPointerType(U)
    end if
    return the type of U
  end if

  if U is a StoreInst then
    O ← first operand of U
    TO ← type of O
    if TO is a PointerType then
      return findPointerType(O)
    end if
    return the type of O
  end if

  if U is a CallInst then
    F ← called function of U
    for each argument A of F do
      if A = Ptr then
        return findPointerType(A)
      end if
    end for
  end if
end for
return nullptr {Type not found}
```

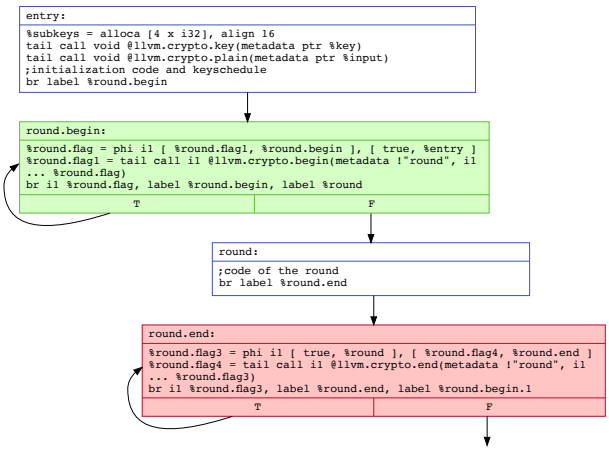


Figure 3.1: Fragment of the CFG of the toy-cipher

- **U is a store instruction**: Similarly to the load instruction case, but with the difference that the store instruction does not return a value and we need to check the type of its first operand instead of the store instruction itself.
- **U is a call instruction**: This is the case in which the pointer is passed as parameter to a different function that might access the memory. So we search the correspondent *argument* of the called function and repeat the search with that argument.

If after having examined all the users of `Ptr` the algorithm is not able to find a memory operation with a valid type, the algorithm terminates yielding a null pointer.

3.1.3. Cipher rounds annotation patch

When Cryptoshield was firstly developed, the information about which round of the cipher an instruction belongs to was obtained through delimiting the code of each round in a different block (delimited by brackets) and annotating each of these blocks with the new pragma annotation `#pragma cipher round`.
Clang was then modified in order to emit the code of the round in a different basic block surrounded by two others basic blocks to delimit the beginning and the end of the round. An example of the first nodes of the CFG produced for the toy-cipher (1.1) is shown in figure 3.1. These two delimiters basic blocks (`round.begin` and `round.end`) were necessary because they contain a loop whose condition was computed by an intrinsic function (respectively `llvm.crypto.begin` and `llvm.crypto.end`) with unknown memory access. In this way Clang was not able to change the order of execution between the instructions before or after the round instructions and the instructions of the round themselves.

The problem in this case arises by the fact that the recent version of Clang to which the

porting was made, is able to reorganize the code, modifying heavily the structure of the CFG compared to what is expected, making the identification of the rounds impossible. The solution to this problem was removing the attribute `always-inline` from the function containing the code of the round, produce the IR, and then reintroduce "manually" the attribute on the IR function and execute only the *always-inline* pass. In this way the instructions of the round remain separated from the others during the first IR generation, and then when function is inlined, no other optimization is performed, so the CFG remain with the correct structure. This operation is performed automatically using CMake, but is just a temporary solution as long as a fix to the problem is found, directly in Clang.

3.2. Dynamic Analysis

The newly developed tool presented in this thesis, as an extension of Cryptoshield (Section 1.5), the tool that allows to perform precise Security Oriented Data Flow Analysis (Section 1.4.2) on cipher algorithms, identify vulnerable instructions, and automatically apply countermeasures, aims to allow new types of *dynamic* analysis to be performed. The tool itself does not produce analysis results, such as which instructions are more vulnerable than others, and its agnostic to the type of code is processing, i.e. the tool is developed to yields better results on cipher algorithms source code, but this is not a requirement, but it generates a library that can be used to compute the intermediate value of each instruction and the *Hamming distance* between two consecutive writes on the same register.

As explained in section 1.3.2 the Hamming distance can be used to produce a model of the power consumption of a cryptographic device in order to predict the power consumption using the hypothesized secret keys. Furthermore, the Hamming distance of interest is the one between two different values written on the same register one after the other because the power consumption correlated with the processed data, is the power consumed during the switching activity of CMOS circuits (Section 1.3.1).

To achieve the goal a set of LLVM passes have been developed. In figure 3.2 is reported the dependency graph of these passes subdivided in *middle-end passes*, i.e. the passes that run during the optimization phase of the compiler, and *backend passes*, which are the passes that need to be performed in the backend phase, only after the register allocation pass (Section 2.1.4).
The passes **IdentifyRoundsPass** and **InputDependenciesPass** are two of the ones already present in Cryptoshield that were ported to the new LLVM version.

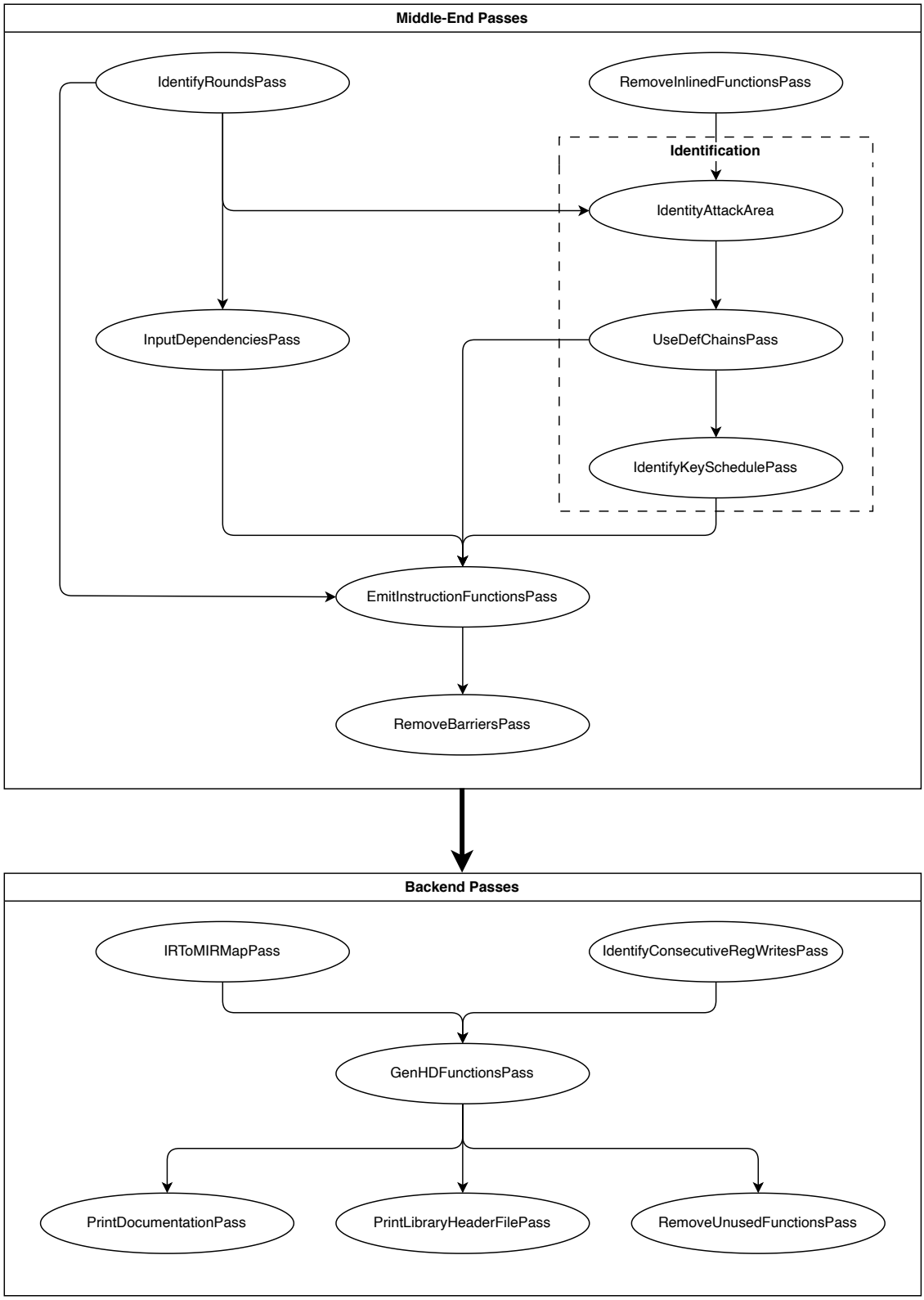


Figure 3.2: Organization of the passes composing the tool

3.2.1. Clang Annotations

The tool exploit the annotations `#pragma cipher round`, `__attribute__((key))` and `__atribute__((plain))`, described in section 1.5.1, already existing in Cryptoshield to enrich the final results with SDFA properties computed statically.
The Clang annotation `#pragma loop unroll` is used on every loop as the tool expects a fully unrolled code, i.e. a code without the presence of loops, in which any loop with static indices is unrolled. Instead, the annotation `__attributes__((always_inline))` is used indirectly as described in section 3.1.3, but in the end the expected code needs to be contained all in one function. Other functions that may be present in the source code but which must not be inlined, should be annotated with `__attributes__((optnone))`.

The new annotation `#pragma attack area` was added to select the portions of the code to consider for generating the library. This annotation works in the same way of the `#pragma cipher round` annotation. During the IR code generation, the portions of the code enclosed in annotated scopes, are emitted in a separate basic block surrounded by two *barriers* basic blocks to prevent the code reordering. This annotation suffers from the same problem highlighted for the `#pragma cipher round` in section 3.1.3.

All the annotations described, except the one to unroll the loops if present, are optional and can be used to improve the results or limit the analysis region.

3.2.2. Middle-End Passes

The set of passes executed during the optimization phase analyze and transform the code in order to satisfy the requirements imposed by the backend passes to work correctly. In particular the at the end of this phase the IR *module* must contain all the functions that computes the intermediates values for each instruction, plus only one single function containing all the cipher code.

The first executed pass, **RemoveInlinedFunctionsPass**, is a module pass whose purpose is to remove all the functions that have been inlined in the cipher function. In particular the pass scan all the functions in the module to search the ones with the attribute *always-inline* or *optnone*, and completely remove them. At the end of this pass only one function must be present in the module, containing all the instructions of the cipher algorithm, including any key schedule instructions that may be present.

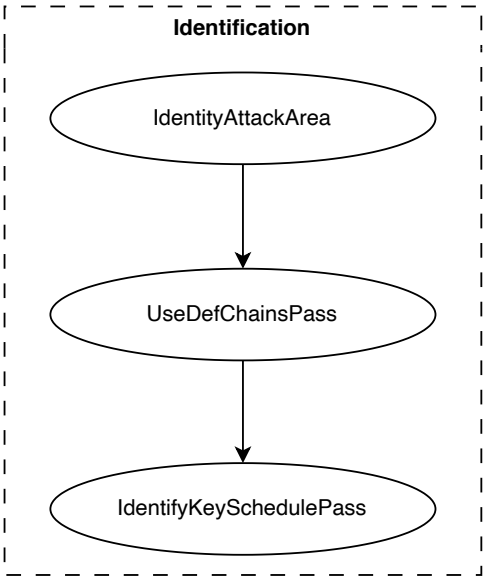


Figure 3.3: Identification phase

Instructions Identification

The next tree passes (**IdentifyAttackArea**, **UseDefChainsPass** and **IdentifyKeySchedulePass**) can be grouped in the *identification phase* (figure 3.3), as roles of these passes is mainly to select and analyze the instructions, obtaining information used by the next pass.

IdentifyAttackArea selects the instructions for which a function for computing the Hamming distance will eventually be present in the library produced by the tool. The pass scans all the instructions searching for the intrinsics functions `crypto_attack_area_begin` and `crypto_attack_area_end`. The instructions between these two functions are selected. If no *attack area* is found, all the instructions are selected. Besides this, a *round-limit* can be passed as command line option to limit the selection of the instructions to a specific round. This option works whether *attack areas* are present or not. To know which round of the cipher each instruction belongs to, the pass exploit the information produced by the analysis *IdentifyRoundsPass* described in section 1.5.

For each of the instructions selected by the previous pass, **UseDefChainsPass** computes the *use-def chain* from the instruction itself to the beginning of the program. The algorithm 3.2 to identify the use-def chains for each instruction works using a work-list queue WL. For each instruction I, WL is initialized with the instruction itself and at each

iteration an instruction C is taken from WL and the operands of I are inserted in WL. When the WL is empty the chain of I is ready and a structure mapping each instruction with its use-def chain is inserted.
In the real implementation there is an optimization that checks if a use-def chain for an operand is already computed, and instead of inserting the operand in the work-list, it copies the use-def chain in the ones of the considered instruction. This optimization speeds up the execution of the analysis when there is a high number of instructions.

The construction of these use-def chains is simple in LLVM IR thanks to the Static Single Assignment form described in section 2.3, and the fact that all the code of the cipher is inside one single function, so in this function there are all the necessary instructions to compute an intermediate value from the beginning of the program, and each intermediate value is defined exactly one time.

Algorithm 3.2 Use-Def Chains Identification

1: $S \leftarrow instructions\ to\ analyze$
2: $CHAINS \leftarrow \varnothing$ {map between instructions and use-def chains}
3: $WL \leftarrow \varnothing$ {the work-list queue}
4: **for each** instruction $I \in S$ **do**
5: $\quad push(WL, I)$
6: $\quad UD \leftarrow \varnothing$ {use-def chain of instruction I}
7: $\quad$ **while** $WL \neq \varnothing$ **do**
8: $\quad\quad C \leftarrow front(WL)$
9: $\quad\quad pop(WL)$
10: $\quad\quad$ **for each** $O \in operands(I) \mid O$ is an Instruction **do**
11: $\quad\quad\quad UD \leftarrow UD \cup \{O\}$
12: $\quad\quad\quad push(WL, O)$
13: $\quad\quad$ **end for**
14: $\quad$ **end while**
15: $\quad CHAINS[I] \leftarrow UD$
16: **end for**

Memory Bridges and Key Schedule Handling

When Clang generate the IR, it has to respect the SSA form, but to simplify the emission of the code in this form, it generates many *memory bridges*. A *memory bridge* is when a value is stored and loaded from memory instead of maintaining it in registers. For example the C function 3.3 is without any optimization is translated to the LLVM IR

shown in 3.4.
Even with a simple function like this, the variable x is first saved in memory, and then loaded in a new temporary variable to be summed and returned. If the algorithm 3.2 is applied to this IR code, the use-def chain of the add instruction (`%add`) will contains only the load (`%0`) and the alloca (`%x.addr`) instructions, missing the store instruction that writes the value of x in memory, since the store is not an operand of any other instruction. Executing the code without the store instruction will produce a random value, as the load instruction will read the content of a portion of memory not initialized.

The majority of these memory bridges are removed with the execution the *mem2reg* pass of the LLVM framework, which promotes memory references to register references [1]. The 3.4 after the *mem2reg* pass is transformed in the 3.5.

Listing 3.3: Simple C function

```c
int function(int x) {
    return x + 1;
}
```

Listing 3.4: LLVM IR generated by Clang without optimizations applied

```llvm
define i32 @function(i32 %x) {
entry:
  %x.addr = alloca i32, align 4
  store i32 %x, ptr %x.addr, align 4
  %0 = load i32, ptr %x.addr, align 4
  %add = add nsw i32 %0, 1
  ret i32 %add
}
```

Listing 3.5: The LLVM IR after the *mem2reg* pass

```llvm
define i32 @function(i32 %x) {
  %add = add nsw i32 %x, 1
  ret i32 %add
}
```

There are, however, some memory bridges that the *mem2reg* pass cannot remove. In a cipher algorithm, the key schedule is one of these. The structure that will contain the subkeys is allocated and then a series of instructions computes and stores the subkeys in that structure.
The **IdentifyKeySchedulePass** analyze the use-def chains computed in the previous pass to identify this possible cases. The algorithm 3.3 works by inspecting each use-def

chain searching for an `alloca` instruction. For each of them, it inspects its users. If the user is a `store` instruction, insert it and its use-def chain in the set of the instructions composing the key schedule, else if it is a `getelementptr` instruction, search for a store using it and put the store and its use-def chain in the set.

The pass constructs a mapping between each `alloca` instruction found and the identified key schedule instructions, and a mapping between each instruction of the use-def chain and the `alloca` instruction that identify the key schedule, if present.

Besides that, if the command line option *opt-size* is active, the same pass emit, for each key schedule found, a new function that takes as parameters the same parameters of the cipher function with the addition of the pointer returned by the `alloca` instruction, and contains all the instructions of the key schedule. The return type is `void` as the function does not compute an intermediate value, but its purpose is to fill the subkey structure.
The generated functions will be used in the next pass to optimize the size of the functions computing the intermediate values.

Intermediate Values Functions Emission

At this point all is all ready to execute the **EmitInstructionFunctionsPass** which for each instruction I in the use-def chains, generate a new function that execute all the instructions in the use-def chain of I, from the beginning of the program and eventually yields its intermediate value.
The intermediate value function (IVF) is built in order to take as parameters the same ones of the cipher function, that are usually the plaintext and the secret key, and has as its last instruction, the one that yields the intermediate value computed by the function that eventually returns it.

The instructions emitted in the IVF are cloned from the ones in the use-def chain of the considered instruction. A *ValueToValueMapTy* is used to keep a reference between the original and the cloned instructions and when their emission is completed, the map is used to set the correct operands for each of the new instruction.
If an instruction has a dependency with a value computed during the key schedule, the instructions identified during the *IdentifyKeySchedulePass* are cloned as well in the same function, keeping the correct execution order.

If present, the metadata of the cipher round and the plaintext and secret key dependencies of the instruction are added to the new generated function, to be used later during the backend passes.
The name of the emitted IVFs is in the form

```
[PREFIX_]i_OPCODE_DEBUGLOC_[ROUND_]ID_id
```

Where `PREFIX` is a string that can be set with the command line option `lib-prefix-opt`, `OPCODE` is the name of the instruction opcode, `DEBUGLOC` is the debug location of the instruction in the form `LINE_COL`, `ROUND`, if present, is the cipher round which the instruction belong and `id` is a number generated for each instruction.

Algorithm 3.3 Key Schedule Identification

1: $KSMap \leftarrow \varnothing$ {Alloca instruction - Key Schedule Mapping}
2: **for each** instruction $I \mid I$ is an AllocaInst **do**
3: $KS \leftarrow \varnothing$
4: **for each** $U \in users(I)$ **do**
5: $S \leftarrow null$
6: **if** U is a StoreInst **then**
7: $S \leftarrow U$
8: **else if** U is a GetElementPtrInst **then**
9: $S \leftarrow findStore(U)$
10: **if** $S \neq null$ **then**
11: $KS \leftarrow KS \cup \{U\}$
12: **end if**
13: **end if**
14: **if** $S = null$ **then**
15: **continue**
16: **end if**
17: $KS \leftarrow KS \cup \{S\}$
18: **if** the first operand O of S is an Instruction **then**
19: $UD_o \leftarrow findUseDef(O)$
20: $KS \leftarrow KS \cup \{O, UD_o\}$
21: **end if**
22: **end for**
23: $KSMap[I] \leftarrow KS$
24: **end for**

With this approach, each intermediate value is computed starting from the inputs, all in a single function. The problem that can arise is that many instructions share most of the use-def chain and key schedule dependency, so cloning each time all the instructions may produce a very large library that cannot be flashed inside the boards used for testing, which have a very limited storage.

For this reason two command line options exists to produce an optimized version of the library. The option *skip-key-schedule* prevents the emission of the IVFs for all the instructions that belongs to the key schedule.
While the most important option is *opt-size* which change completely the way the IVFs are emitted. Instead of cloning all the instructions of the use-def chain, the pass maintain a map between the instructions and the IVFs already emitted, and when a new IVF has to be generated, the algorithm search the IVFs that computes the operands of the intermediate value to be computed. If these functions exists the new IVF is built calling the IVFs of the operands and using the results as the new operators for the instruction. An example of an IVF emitted for the toy-cipher 1.1, is shown in 3.6 for the non-optimized version and 3.7 for the optimized one.

Listing 3.6: Example of non-optimized IVF of the toy-cipher

```
1 define i32 @tc_i_xor_18_14_Round_0_ID_13(ptr %key, ptr %input, ptr ↩
    %output) #5 {
2 entry:
3   %subkeys = alloca [4 x i32], align 4
4   %0 = load i32, ptr %input, align 4
5   %arrayidx.i = getelementptr inbounds i32, ptr %key, i32 1
6   %1 = load i32, ptr %arrayidx.i, align 4
7   store i32 %1, ptr %subkeys, align 4
8   %2 = load i32, ptr %key, align 4
9   %arrayidx3.i = getelementptr inbounds i32, ptr %subkeys, i32 1
10  store i32 %2, ptr %arrayidx3.i, align 4
11  %xor.i = xor i32 %2, %1
12  %arrayidx6.i = getelementptr inbounds i32, ptr %subkeys, i32 2
13  store i32 %xor.i, ptr %arrayidx6.i, align 4
14  %shl.i = shl i32 %1, 10
15  %xor8.i = xor i32 %shl.i, 64751
16  %arrayidx9.i = getelementptr inbounds i32, ptr %subkeys, i32 3
17  store i32 %xor8.i, ptr %arrayidx9.i, align 4
18  %3 = load i32, ptr %subkeys, align 4
19  %xor.i14 = xor i32 %0, %3
20  ret i32 %xor.i14
21 }
```

Listing 3.7: The optimized version of the IVF 3.6. The parameters of the call instructions are omitted.

```
define i32 @tc_i_xor_18_14_Round_0_ID_9(ptr %key, ptr %input, ptr ↩
    %output) #5 {
entry:
  %0 = call i32 @tc_i_load_29_20_ID_2(...)
  %1 = call i32 @tc_i_load_18_17_Round_0_ID_8(...)
  %2 = xor i32 %0, %1
  ret i32 %2
}
```

There are tree cases that require particular attention when the optimized version of the IVF is generated.

The first case is when one of the operands of the instruction is an `alloca` instruction, which means that the instruction use directly a value computed in the key schedule. For this operand instead of calling a previous generated IVF, the `alloca` instruction is cloned and used as new operand, and the key-schedule function generated during the *IdentifyKeySchedulePass* is called to fill the structure.

The second case is when the instruction is a `load` that has as operand a GEP instruction which computes an address to access the subkeys structure. In this case, calling the IVF of the GEP would not work, because even if the called IVF is calling the function for the key schedule, the structure for the subkeys is allocated in the GEP IVF, and the `alloca` instruction, allocates the memory in the stack of a function. When the GEP IVF return with the computed address, the subkeys structure is not allocated anymore and the load may yield an incorrect value.
This problem is solved, not emitting the IVF for the GEP instructions, but instead emit the GEP and the load in the same function that computes also the subkeys.

The third case, which however is not yet handled, is when an IVF for a key schedule function has to be generated. These instructions access and *writes* to the subkeys structure when it is not ready yet. Special functions, similar to the IVF but that takes as additional parameter the pointer to the subkey structure, has to be generated to handle this case. Currently, this case is handled by emitting the non-optimized version of the function.

Barriers Removal

The last pass executed in the optimization phase is the **RemoveBarriersPass**, which the only purpose is to remove the basic blocks delimiting the rounds and the attack areas,

and merge all the remaining basic blocks in one single basic block. This pass is necessary to simplify the subsequent analyses and to remove the intrinsics functions that LLVM is not able to translate in Machine Intermediate Representation (MIR).

3.2.3. Backend Passes

Once all the IVFs have been generated, the *Hamming distance* calculation is straightforward, in fact given two intermediate values v_1 and v_2, their Hamming distance is the number of bits that differ from v_1 and v_2 and can be computed as the *Hamming weight*, i.e. the number of bits set to one, of $v_1 \oplus v_2$.
There is however one missing information before the functions computing the Hamming distance can be generated. The Hamming distance we are interested in, is not the one between two random intermediate values, but the one between two values written in the same register one after the other.
To gain this information all the subsequent passes have to be performed during the backend phase of the compiler, and more precisely only after that the physical registers have been chosen for each instruction, in the register allocation pass.

Register Writes Identification

In the backend phase, the IR is translated in the MIR (Machine Intermediate Representation), which is no longer independent of the target machine, and in which each instruction reads and writes on registers that initially are *virtual* and of infinite number, similarly to the IR, but after the register allocation pass they become *physical* and correspond to the real registers available on the target architecture.

The pass **IdentifyConsecutiveRegWritesPass** scan all the instructions in execution order and build a map between the register and the instructions writing in that register.

Algorithm 3.4 Register Writes Identification

1: $RegMap \leftarrow \varnothing$
2: **for each** instruction I in execution order **do**
3: $\quad O \leftarrow operands(I)[0]$
4: $\quad$ **if** O is a definition **then**
5: $\quad\quad RegMap[O] \leftarrow RegMap[O] \cup \{I\}$
6: $\quad$ **end if**
7: **end for**

IR to MIR Mapping

The information about the consecutive register writes is associated with the MIR instructions, but the IVFs are associated with the IR instructions. After that the IR is translated in MIR the two representations are almost completely independent and there is no direct way to trace from which IR instruction each MIR instruction was generated, because from a single IR instruction, more than one MIR instruction can be emitted, based on the architecture capabilities. Also, the metadata associated with an IR instruction are lost during this translation.

One information that is kept during the generation of the MIR is the *debug location*. This information is associated to each instruction and contains the line and the column of the source file from where the instruction was originally generated.
The pass **IRToMIRMapPass** exploit this information to build a mapping between each MIR instruction and the respective IR instruction.

The algorithm 3.5 first create an array P of pairs that associate each MIR instruction with a string s built from their debug location, then scan the IR instructions to build a map M that use as keys strings of the same format of s and as values stacks of IR instructions associated with that location. A stack is necessary because there can be more than one IR instruction associated with the same debug location, e.g. the instructions generated from an unrolled loop have all the same location.
Eventually the array P is traversed backwards and, for each pair, if the location string also exists in the map M, an IR instruction is popped from the corresponded stack and inserted in a new map between the MIR instructions and the IR instructions.

Hamming Distance Functions Generation

The last important step is emission of the functions that computes the Hamming distance and yields other data for the instructions that can be used to analyze the cipher algorithm. This step is performed by the **GenHDFunctionsPass** pass. Exploiting the analyses performed by the two previous passes, the emission of these functions is straightforward.

The pass iterate over the registers and for each one iterate pairwise the machine instructions that writes to it consecutively. If for both the machine instructions there is a mapped IR instruction and an IVF, the Hamming distance function can be emitted, otherwise the generation of this pair of instruction is skipped.

Algorithm 3.5 MIR to IR Instructions Mapping

```
1:  P ← ∅
2:  for each machine instruction MI do
3:      if MI has a debug location then
4:          s ← getDebugLocationString(MI)
5:          P ← P ∪ {(s, MI)}
6:      end if
7:  end for
8:  M ← ∅
9:  for each IR instruction I do
10:     if I has a debug location then
11:         s ← getDebugLocationString(I)
12:         push(M[s], I)
13:     end if
14: end for
15: InstrMap ← ∅
16: for each (s, MI) ∈ P iterated backwards do
17:     if s ∈ M then
18:         I ← pop(M[s])
19:         InstrMap[MI] ← I
20:     end if
21: end for
```

The generated function takes as parameters the same of the cipher algorithm, plus a pointer to a `struct` in which, at runtime, the generated data, i.e. the involved register, the intermediate values yielded by the two instructions, the Hamming distance between the values, the execution time of the two IVFs and other information obtained during the static analyses performed by the Cryptoshield passes, are saved.
The function call one after the other, the two IVFs, and computes the Hamming distance by first xoring the two results and then calling the intrinsic function `llvm.intrinsic.ctpop` which counts the number of bits set.
Before and after each IVF call, the function `getTime`, that has to be defined externally, is called to compute the execution time for the two IVFs.
The data about the cipher round or secret key and plaintext dependencies are retrieved from the metadata that was associated on each IVF during the *EmitInstructionFunctionsPass* pass.

Header File and Documentation Generation

The previous pass, saves all the information of the generated functions, which is then exploited by the two passes, **PrintDocumentationPass** and **PrintLibraryHeaderFilePass**.

The first pass is very simple. It generates two equivalent files, one in text format and the other in CSV format, containing for each generated functions the information about the involved register, the two instructions and their respective location in the source code and the name of the Hamming distance function.

The second pass is more useful, as simplify the use of the library, by generating a C header file (.h) containing the required inclusions, the definition of the struct `instr_data_struct` used to collect the data and the declaration of all the Hamming distance functions. Each declaration is also decorated with a comment with the name of the two IVF called by the Hamming distance function, the involved register and the secret key and plaintext dependencies.
Furthermore, a utility function `getRawData` is generated. This function takes as parameter an array of `instr_data_struct` and the parameter of the cipher algorithm, and automatically calls all the Hamming distance functions generated.
The pass contains a simple mechanism to convert the LLVM IR types to their C types counterparts.

The header file generated can be used to include the library in another C program to call all the functions contained in the assembly file emitted by LLVM after the code generation.

Final Optimization

If the command line option `keep-unused` is **not** enabled, the last **RemoveUnusedFunctionsPass** pass is executed.
The pass searches for all the IVFs that were ultimately not used by any Hamming distance function, and removes it. The pass helps to optimize the size of the generated assembly file, since many generated IVFs associated with IR instructions that were eventually not mapped to any MIR instruction, remain unused. The pseudocode of the algorithm is shown in 3.6

Algorithm 3.6 Unused Functions Removal

```
1: getCallsOf(f):
2: C ← ∅ {Calls in f}
3: for each instruction I ∈ f do
4:    if I is a CallInst then
5:       C_I ← getCallsOf(I)
6:       C ← C ∪ {I, C_I}
7:    end if
8: end for
9: return C
10:
11: U ← ∅ {Used functions}
12: for each HD function f_HD do
13:    f_IVF1 ← f_HD.firstIVF
14:    f_IVF2 ← f_HD.secondIVF
15:    C_1 ← getCallsOf(f_IVF1)
16:    C_2 ← getCallsOf(f_IVF2)
17:    U ← U ∪ {f_HD, f_IVF1, f_IVF2, C_1, C_2}
18: end for
19: for each function f in module do
20:    if f ∈ U ∧ f ≠ intrinsic ∧ name(f) ≠ getTime then
21:       removeFunction(f)
22:    end if
23: end for
```

4 | Validation

In this final chapter is described how the tool presented in the previous chapter (3) was tested.
The tests were performed on an embedded device based on the ARMv7 architecture described in section 4.1. Various ciphers, either historically relevant or still important today, as described in section 4.2, was analyzed, and the plots of the Hamming distances generated from the collected data for each cipher, was reported is section 4.3.

To automate the test of many different ciphers, a python script was developed. The script, for each tested cipher, is able to **compile** the program for the embedded device, selecting the right cipher by setting the `CIPHER` compilation flag (e.g. To select AES128, the flag `-DCIPHER=AES128` has to be set), **flash** the binary file on the embedded device using the `st-flash` tool, and finally open a connection with the device to **read** and **elaborate** the data produced by the device. The results are stored raw in a semicolon separated CSV file and in an XLSX file containing the table and the chart of the Hamming distances. An SVG version of the chart is also produced.

Before analyzing the cryptographic algorithms however, the tool was verified, that means that the results of the Hamming distance functions was checked to be correct. This was done using a simple test program that staring from a specific value, perform a bunch of operations, such us shifting operations, one after the other, in such a way that all the intermediate values of the program were known, and so also the all the Hamming distances between each pair of instruction.
The test program has to be simple enough and has to be compiled without any optimization, in order to use only one register and keep the original instruction order. The library produced by the tool was executed on the test board, as any other tested algorithm, and the results of each Hamming distance function was compared to the precomputed ones.

4.1. Testing Environment

The embedded device used to test the tool and collect the data of the cipher was the STM32F411E-DISCO board (figure 4.1), featuring an ARMv7 32-bit Cortex-M4 CPU, 512 Kbytes of flash memory and 128 Kbytes of RAM [41].
The Cortex-M4 CPU execute the Thumb instruction set, which is a subset of the traditional 32 bits ARM instructions, but, they are 16 bits long and so the produced code is more compact. During the tests, the CPU was configured to operate at 72 MHz.

The `getTime()` function used by the Hamming distance functions is implemented to return a counter produced by a timer operating at 72 MHz. The time in the raw data retrieved from the board is the difference between the value of the counter after the execution of the function and the one before the execution of the function.

The data is transmitted through the user USB port (CN5). For convenience the `_write` function was implemented to write on the USB port when available, as showed in the code 4.1, and this enables the possibility of using the `printf` standard function to format and transmit the data automatically.

Listing 4.1: Implementation of the _write function

```
int _write(int file, char *ptr, int len) {
    static uint8_t rc = USBD_OK;

    do {
        rc = CDC_Transmit_FS(ptr, len);
    } while (USBD_BUSY == rc);

    if (USBD_FAIL == rc) {
        // Error!
        return 0;
    }
    return len;
}
```

4.2. Ciphers Analyzed

All the cipher analyzed are block ciphers whose implementations was modified in order to be fully unrolled and with the addition of the custom clang attributes to identify the key, the plaintext and the rounds. In the table 4.1 there is a summary of the analyzed ciphers with their features, as the key size or the internal structure.

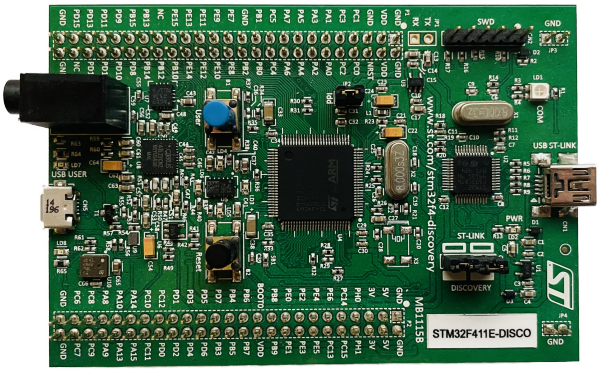


Figure 4.1: STM32F411E-DISCO board that was used.

Cipher	Structure	Key size	Block size	Rounds	Year
AES	SPN	128, 192, 256	128	10, 12, 14	2002
DES	Feistel	56	64	16	1977
TDES2	Feistel	112	64	32 $(2 * 16)$	1978
TDES3	Feistel	168	64	48 $(3 * 16)$	1978
CAST	Feistel	128, 256	64, 128	12, 16, 48	1996
CLEFIA	Feistel	128, 192, 256	128	18, 22, 26	2007
HIGHT	Feistel	128	64	32	2006
MISTY1	Nested Feistel	128	64	8	1995
SEED	Nested Feistel	128	128	16	1998
Serpent	SPN	128, 192, 256	128	32	1998
Simon	Feistel	128, 192, 256	128	68, 69, 72	2013
Speck	ARX	128, 192, 256	128	32, 33, 34	2013
XTEA	Feistel	128	64	64	1997

Table 4.1: Main characteristics of the analyzed ciphers.

For some cipher taken in consideration it was not possible to obtain data with different reason:

- **CLEFIA**: The cipher was not tested at all, because as implemented, is not possible to unroll every loop in the code, and so the used implementation is not compatible with the current version of the analysis tool.
- **CAST and Serpent**: For both of these ciphers the tool runs correctly, but the board is not able to terminate the execution in a reasonable time. Further clar-

ifications of why this happens needs to take place, to understand if the cause of this behavior is related to a bug in the developed tool, or if new optimizations are needed to speed up the final execution.

- **TDES2 and TDES3**: Also for both of these ciphers the tool runs correctly, but in this case the produced binary file is too large and does not fit in the 512 KB of flash memory of the testing board described in section 4.1.

4.2.1. AES

Rijndael, the original name of the cipher, was developed in 1997 as a candidate for becoming the new encryption standard and replace DES. It finally became the new standard in 2002 with the name of Advanced Encryption Standard (AES) [30].
The cipher is structured as a Substitution-Permutation Network (SPN). The length of the input block, the output block and state block is 128 bit. The cipher key has a variable size and can be 128, 192 or 256 bit long, which determine also the number of rounds, that are respectively 10, 12 or 14 [34].
AES can be efficiently implemented both in software and in hardware, and it works well also in embedded device such as smart cards [30].

4.2.2. DES

The algorithm was originally developed by IBM, and became the encryption standard, with the name of Data Encryption Standard (DES), in 1977, becoming the most widely used encryption algorithm worldwide [43].
The cipher operates on 64 bit data blocks using a cipher key of 56 bit, performing 16 rounds of the Feistel network, i.e. a series of substitution and permutation primitives, applied iteratively to the data blocks.
Due to the short cipher key, the algorithm was successfully attacked multiple times, using a brute-force attack, and new variants of the cipher was developed to increase the security [33].

The two new variants, known as "Triple-DES", are the **TDES2** and **TDES3** which applies the DES algorithm 3 times on the same data block, using respectively 2 and 3 different keys. In particular the TDES2 use a key of 112 bit ($2 * 56$) and during third application of DES, reuse the first key, while the TDES3 use a key of 168 bit ($3 * 56$), and so use a different key for each application of the DES algorithm.

Despite the new variants DES was replaced by AES, described in section 4.2.1, that

became the new encryption standard in 2002.

4.2.3. CAST

Designed by Carlisle Adams and Stafford Tavares in 1996, CAST-128, also known as CAST5, is a DES-like algorithm based on a Feistel network that have a good resistance to differential cryptanalysis [37].
The cipher operates on 64 bit blocks and allow key size that vary from 40 bit to 128 bit in 8-bit increments.
For key sizes up to 80 bit (included) the cipher perform 12 rounds, while for key sizes greater than 80 bit the cipher perform 16 rounds [4].

A new version of the algorithm was proposed in 1997 to become the new encryption standard and replace DES, but the competition was finally won by Rijndael, described in section 4.2.1.
The new version called of the algorithm, known as CAST-256 or CAST6, has a similar structure to CAST-128, but it operates on blocks of double size (128 bit) and uses a variable key size up to 256 bit, performing 48 rounds [5].

4.2.4. CLEFIA

Developed by Sony in 2007 to protect the copyright of the digital distribution, such as music or images, CLEFIA operates on 128 bit blocks and support key lengths of 128, 192 or 256 bits. The algorithm is structured as a Feistel network with 18, 22 or 26 rounds depending on the key size, and can be implemented both in hardware and software efficiently [39].

4.2.5. HIGHT

Described by Deukjo Hong et al. in [24], HIGHT (high security and light weight) is a block cipher designed specifically to be implemented in low resource hardware, like RFIDs (Radio Frequency Identification).
The cipher has a Feistel network structure with 32 rounds, and operates on 64 bit blocks using a secret key of 128 bits.
The cipher can be easily implemented on 8-bit CPU where it performs better than AES.

4.2.6. MISTY1

Designed by Matsui in 1997 for Mitsubishi Electric Corporation, MISTY1 was included in the European NESSIE-recommended cipher [15].
The cipher has a nested Feistel network structure and operates on 64 bit blocks with a 128 bit key. The number of rounds is variable, but 8 rounds are recommended [31].
MISTY1 was successfully attacked in 2015 [15]. The attack is infeasible because requires a complete codebook of $2^{69.5}$ chosen plaintext to recover the full key, but it proves that the effective safety of MISTY1 is of 2^{70}, significantly less than what was considered before.

4.2.7. SEED

SEED was developed South Korea in 1998 by KISA (Korea Information Security Agency), to substitute the old vulnerable ciphers, and became a standard in the country.
The algorithm is described in RFC 4269 [44] and has a Feistel structure with 16 rounds, and operates on 128 bit blocks using a 128 bit key.

4.2.8. Serpent

Designed and described in [13] by Ross Anderson et al. in 1998, Serpent was a candidate to become the new Advanced Encryption Standard, and ranked second to Rijndael due to the many rounds that slow down its execution, despite having a greater security margin than Rijndael.
The cipher has 32 rounds of a substitution-permutation network (SPN) and operates on four 32-bit words composing the block of 128 bits. The key can be 128, 192 or 256 bits long, and all the keys smaller than 256 bits are mapped to a 256 bits key adding a "1" bit to the most significant bit (MSB), followed by as many "0" bits as required to obtain a 256 bits key. In this way the cipher can support also smaller key if required.

4.2.9. Simon and Speck

Designed by NSA (National Security Agency) in 2013 [17], Speck and Simon are two families of block ciphers designed to be *lightweight*, highly-optimized and flexible, i.e. they have been developed to be used on embedded environments, but compared to other ciphers, they are not optimized for a specific platform but performs well across different platforms.
Both Simon and Speck performs well on hardware and software, but Simon has been optimized for performance on hardware device, while Speck for performance in software.

Both ciphers operate on blocks size that range from 32 to 128 bits using a variable size key from 64 to 256 bits. Simon has a Feistel structure optimized to be extremely small in hardware, while Speck is an ARX (Add-Rotate-Xor) cipher. All the possible configurations of both ciphers are reported in table 4.2, but only the 128-bit block size configurations were considered for analysis in this thesis.

Block size	Key size	Simon rounds	Speck rounds
32	64	32	22
48	72	36	22
	96	36	23
64	96	42	26
	128	44	27
96	96	52	28
	144	54	29
128	128	68	32
	192	69	33
	256	72	34

Table 4.2: All the possible configurations of Speck and Simon ciphers.

4.2.10. XTEA

Designed in 1997 by Roger Needham and David Wheeler, XTEA is the eXtension of the TEA cipher [42]. The cipher was designed to have a very simple round function but to perform a large number of iterations to make it secure. The algorithm is a Feistel structure that operates on 64 bit blocks using a 128 bit key. The number of rounds is variable but is suggested to perform at least 64 rounds.

4.3. Results

In this section the results from the analyzed ciphers are presented. Table 4.3 reports for each cipher the parameters used to analyze it, that are the number of rounds were analyzed and whether the optimization described in section 3.2.2, aimed at reducing the binary file size, was active or not, and the number of Hamming distance functions were finally emitted.

The optimization does not interfere with the intermediate values and so with the Hamming

distances computed, but in the full data, that are not reported here, is possible to notice that the execution time on the size-optimized versions is much larger as many functions has to be called to compute a single intermediate value instead of just one single function in the non size-optimized versions.

The plots visualized here are constructed using the python library *matplotlib*, and represent the value of the Hamming distance for all the emitted functions, sorted by the machine instruction execution order. The colors on the charts represents the dependencies of the instructions on the plaintext or/and on the secret key.

The collected data have shown known characteristics of ciphers, and in particular is possible to notice that:

- The three variants of AES (figures 4.2, 4.3 and 4.4) show and increase in the length of the key schedule part, i.e. the part of the plot only dependent on the key, as the key size increases.
- DES (figure 4.5) has a very large key schedule compared to the cipher rounds and a very compact loading of the plaintext. In the key schedule part it is also possible to observe several repeated peaks that represent the loading of different portions of the key.
- MISTY1 (figure 4.7) has a simple key schedule. In fact, even though only 3 of the 8 rounds of the cipher were analyzed, they occupy a large portion of the graph.
- With Simon (figures 4.9, 4.10 and 4.11), the tool produced only a few Hamming distance functions compared to the other ciphers. The cause is that the implementation of the cipher work with 64-bit integers, but the board used to test, described in section 4.1, has a 32-bit architecture. During the compilation, the machine code produced is very different from the intermediate representation and the mapping between the IR instructions and the MIR instructions was imprecise. Even the result of the mapped instructions may be wrong because a single MIR instruction that produces a 32-bit value may be mapped with an IR instruction that computes the whole 64-bit intermediate value. Besides that, the plots still show the execution of the key schedule between each of the 9 rounds analyzed.
- Speck (figures 4.12, 4.13 and 4.14) should have a structure, similarly to Simon, where the key schedule is performed between each round. In the graph however, it seems that only 2 rounds were performed instead of 3. This can be explained considering that during the emission of the machine code, the instructions are sometimes rescheduled to optimize the execution, and the graphs show the values in the real

execution order. The key schedule for the second round was anticipated and performed together with the one of the first round.
As for AES is possible to observe the growth of the key schedule as the key size increases.

- XTEA (figure 4.15) is a very lightweight cipher, and as expected, it was possible to analyze a much larger number of rounds compared to the other ciphers analyzed.

Cipher	Key size	Rounds analyzed	Optimized	HD functions
AES	128	3	Y	292
	192	3	Y	262
	256	3	Y	327
DES	56	3	Y	1158
HIGHT	128	3	Y	292
MISTY1	128	3	Y	239
SEED	128	3	N	373
SIMON	128	9	Y	54
	192	9	Y	54
	256	9	Y	55
SPECK	128	3	N	161
	192	3	N	163
	256	3	N	166
XTEA	128	16	N	170

Table 4.3: Parameters used to test each cipher, and number of Hamming distance functions generated.

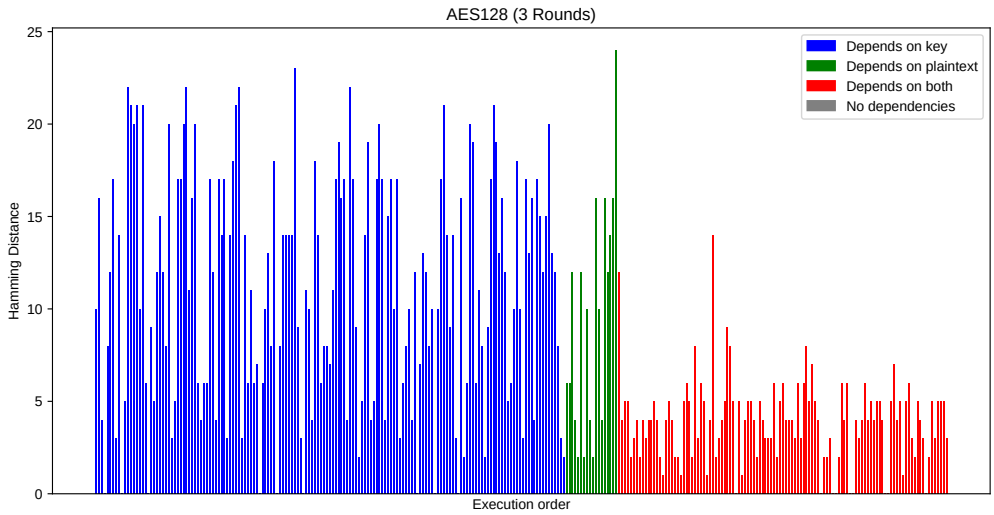


Figure 4.2: Plot of hamming distances computed for the first three rounds of AES128.

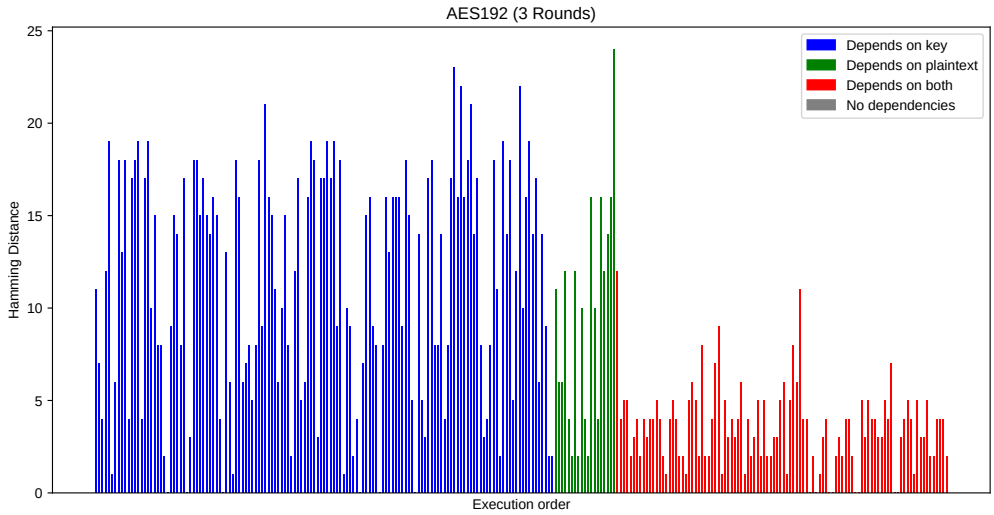


Figure 4.3: Plot of hamming distances computed for the first three rounds of AES192.

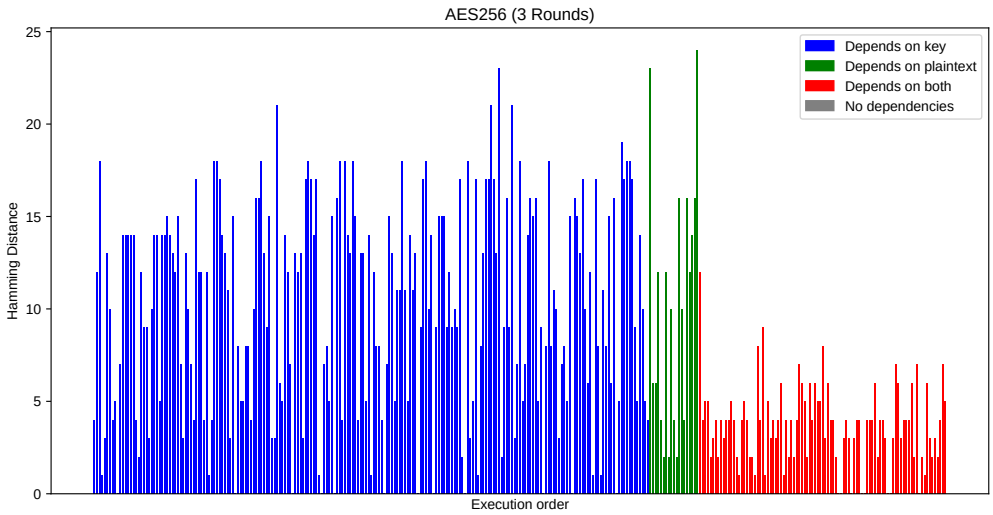


Figure 4.4: Plot of hamming distances computed for the first three rounds of AES256.

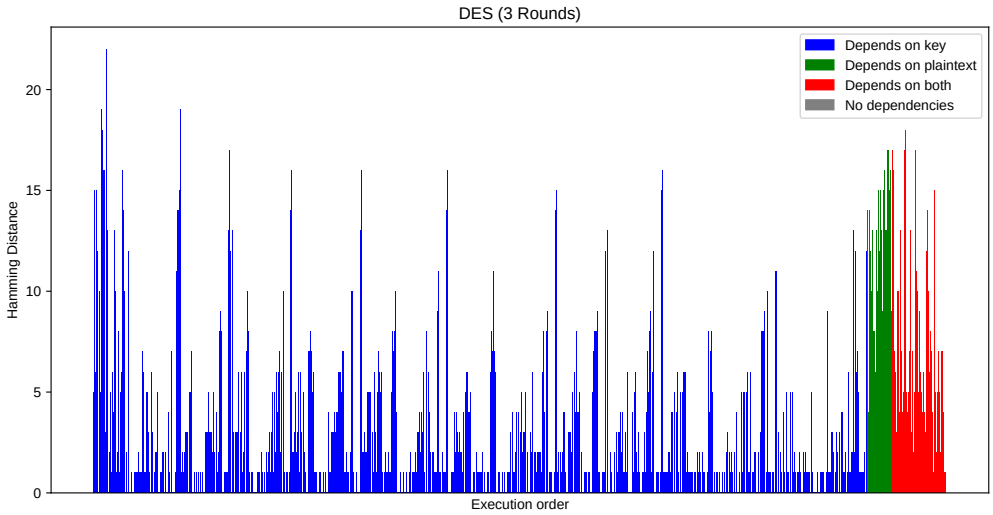


Figure 4.5: Plot of hamming distances computed for the first three rounds of DES.

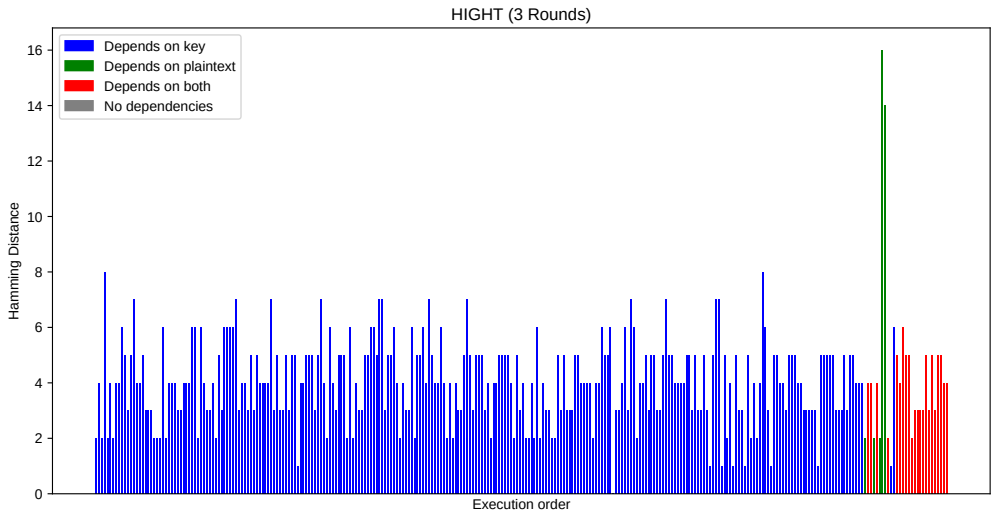


Figure 4.6: Plot of hamming distances computed for the first three rounds of HIGHT.

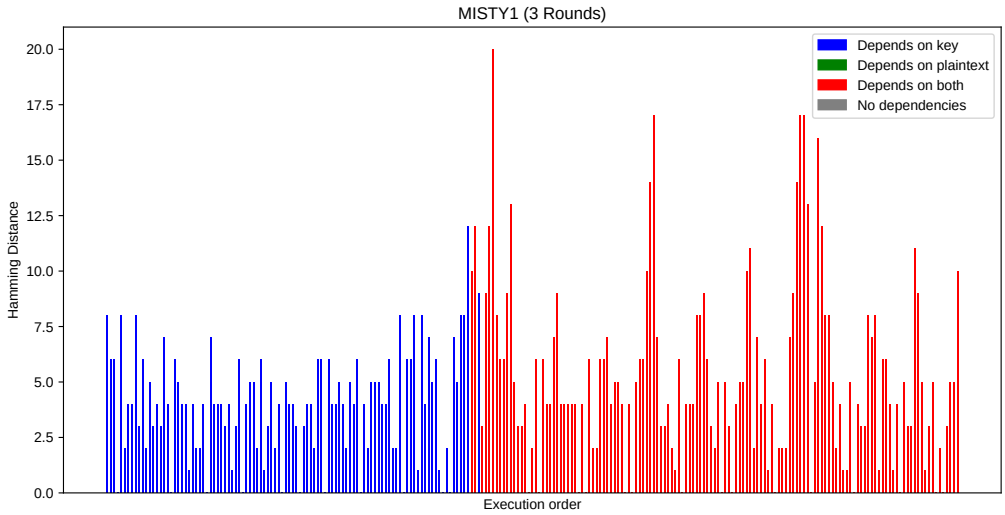


Figure 4.7: Plot of hamming distances computed for the first three rounds of MISTY1.

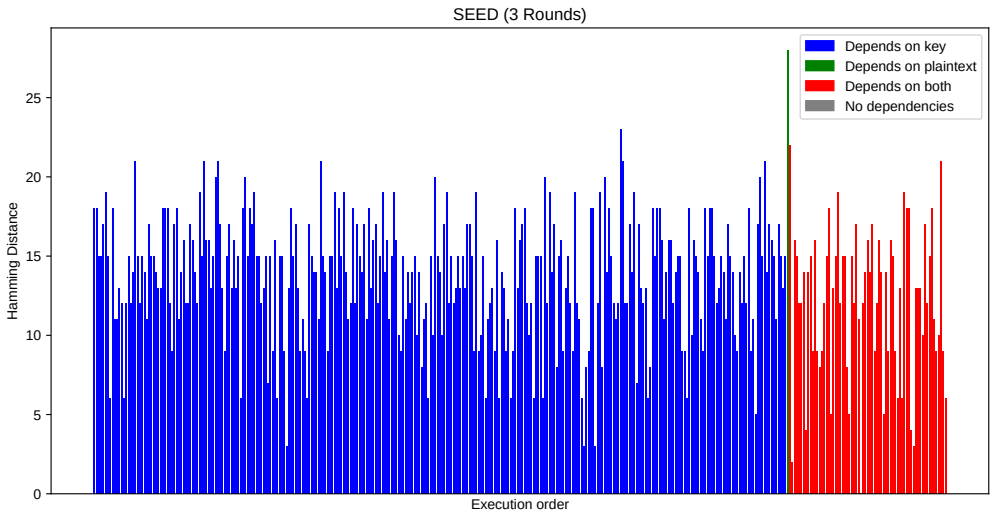


Figure 4.8: Plot of hamming distances computed for the first three rounds of SEED.

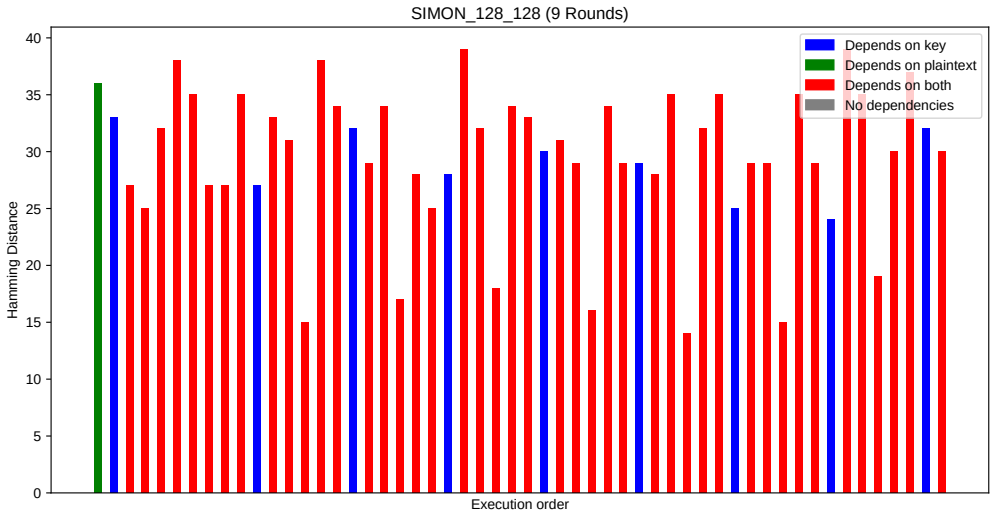


Figure 4.9: Plot of hamming distances computed for the first nine rounds of SIMON128.

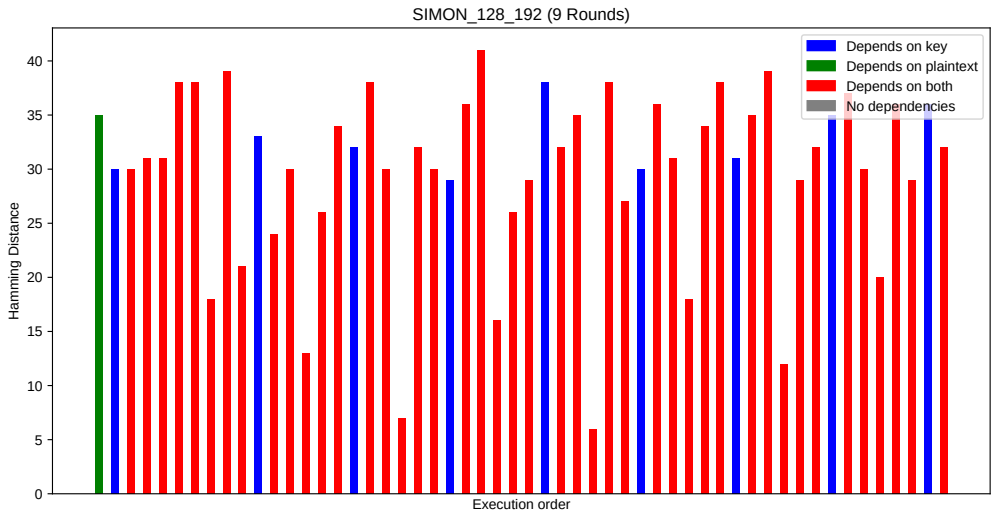


Figure 4.10: Plot of hamming distances computed for the first nine rounds of SIMON192.

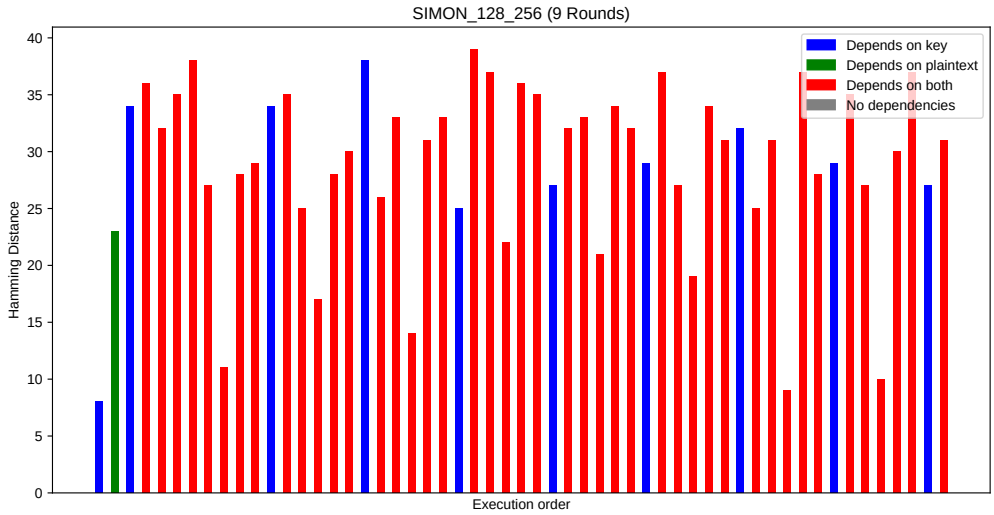


Figure 4.11: Plot of hamming distances computed for the first nine rounds of SIMON256.

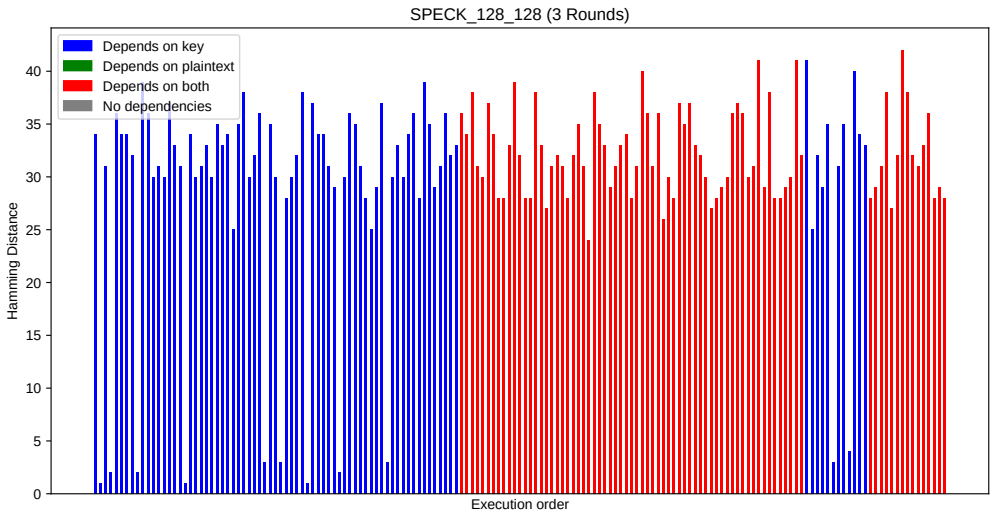


Figure 4.12: Plot of hamming distances computed for the first three rounds of SPECK128.

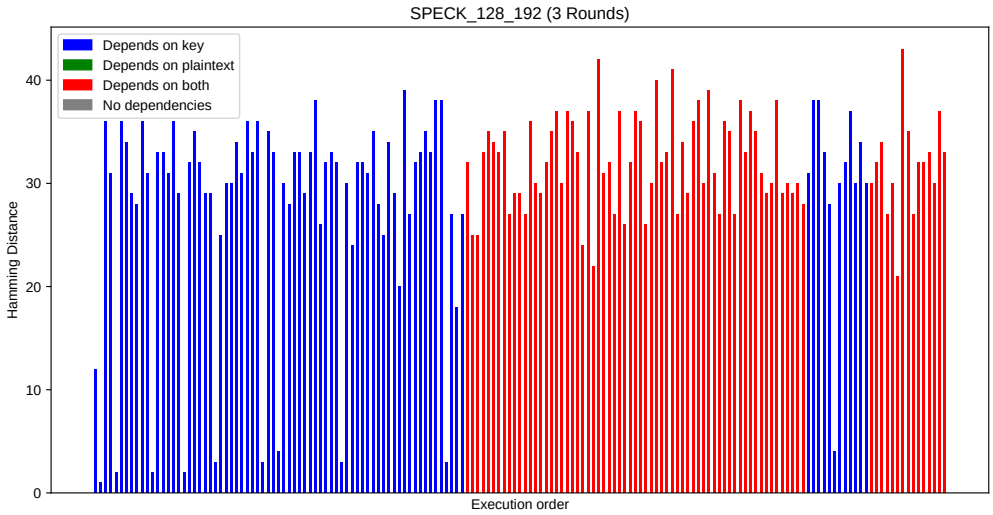


Figure 4.13: Plot of hamming distances computed for the first three rounds of SPECK192.

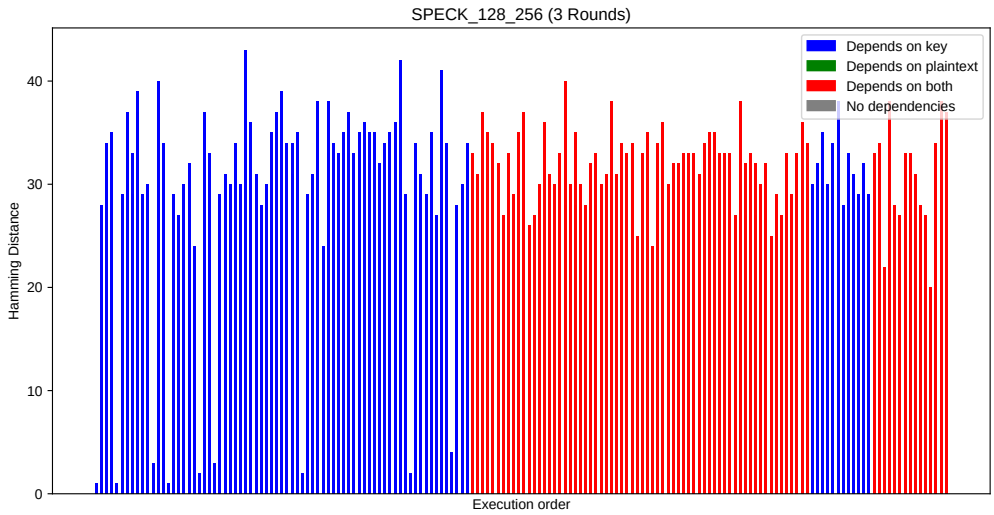


Figure 4.14: Plot of hamming distances computed for the first three rounds of SPECK256.

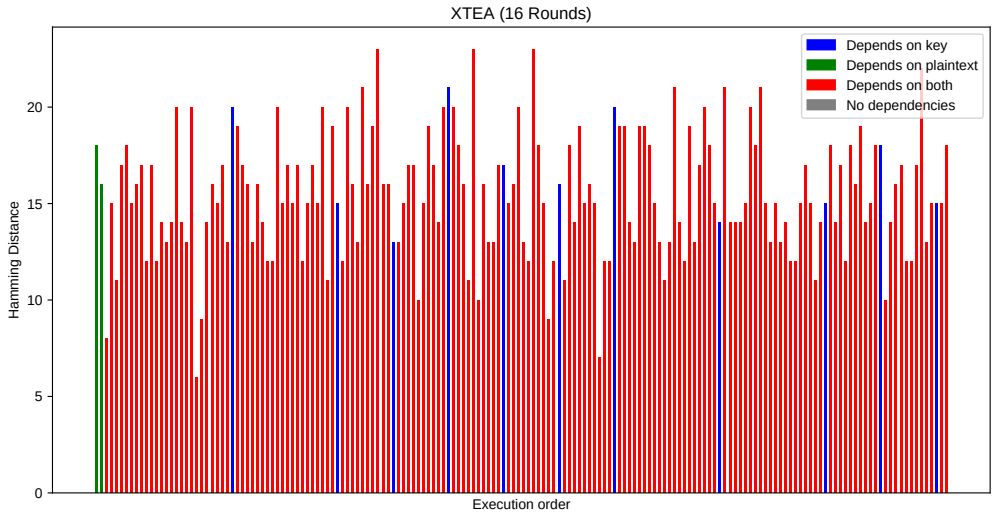


Figure 4.15: Plot of hamming distances computed for the sixteen three rounds of XTEA.

5 Conclusions

In this thesis I described how I ported to a recent version of the LLVM framework, "Cryptoshield", a tool that exploiting the concept of Security-Oriented Data Flow Analysis (SDFA), described in section 1.4.2, performs set of automated in-depth analyses to find the characteristics of cryptographic algorithms, computing a vulnerability index for each instruction of the cipher and based on this index, it automatically applies countermeasure against side-channel attacks.

I then extended the work proposing a way to dynamically analyze the ciphers, computing a model of the Hamming distance calculated between each pair of intermediate values consecutively written in the same memory location (register).
In order to know in which register a value will be written, the tool needs to operate in two different phases of the compiler pipeline. The first part operates in the optimization phase because it needs to access to the intermediate representation to identify the intermediate values and generate the functions that compute them, while the second part has to operate in backend phase of the compiler because it must have access to the real machine instructions and most importantly to the physical registers assigned during the register allocation pass. The result produced by the tool for each cipher, is a library containing all the functions that can be used to compute the Hamming distances of interest.
The tool was used on 11 cipher algorithms in different variants and key sizes, and the generated libraries was used on a test board to produce the plots described in chapter 4.

The work although applied to the field of cryptography, makes extensive use of the techniques and principles used by compilers, which prove to be a fundamental tool for ensuring the security of cryptographic algorithms, improving the efficiency and precision of the analyses and reducing the possible errors generated by a manual application of countermeasures.

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

Bibliography

[1] LLVM Documentation, . URL https://llvm.org/docs/.

[2] Opaque Pointers — LLVM 20.0.0git documentation, . URL https://llvm.org/docs/OpaquePointers.html.

[3] LLVM. In A. Brown, G. Wilson, and C. Lattner, editors, *The architecture of open source applications: elegance, evolution, and a few fearless hacks*. s.l., 2011. ISBN 978-1-257-63801-7.

[4] C. Adams. The CAST-128 Encryption Algorithm. Request for Comments RFC 2144, Internet Engineering Task Force, May 1997. URL https://datatracker.ietf.org/doc/rfc2144. Num Pages: 15.

[5] C. Adams, H. Heys, S. Tavares, and M. Wiener. An analysis of the CAST-256 cipher. In *Engineering Solutions for the Next Millennium. 1999 IEEE Canadian Conference on Electrical and Computer Engineering (Cat. No.99TH8411)*, volume 1, pages 361–366 vol.1, May 1999. doi: 10.1109/CCECE.1999.807225. URL https://ieeexplore.ieee.org/document/807225. ISSN: 0840-7789.

[6] G. Agosta, A. Barenghi, and G. Pelosi. A code morphing methodology to automate power analysis countermeasures. In *Proceedings of the 49th Annual Design Automation Conference*, pages 77–82, San Francisco California, June 2012. ACM. ISBN 978-1-4503-1199-1. doi: 10.1145/2228360.2228376. URL https://dl.acm.org/doi/10.1145/2228360.2228376.

[7] G. Agosta, A. Barenghi, M. Maggi, and G. 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. ISBN 978-1-4503-2071-9. doi: 10.1145/2463209.2488833. URL https://dl.acm.org/doi/10.1145/2463209.2488833.

[8] G. Agosta, A. Barenghi, G. Pelosi, and M. Scandale. The MEET Approach: Securing Cryptographic Embedded Software Against Side Channel Attacks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 34(8):1320–

1333, Aug. 2015. ISSN 0278-0070, 1937-4151. doi: 10.1109/TCAD.2015.2430320. URL `http://ieeexplore.ieee.org/document/7103025/`.

[9] G. Agosta, A. Barenghi, and G. 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. ISBN 978-1-4503-4128-8. doi: 10.1145/2903150.2911707. URL `https://dl.acm.org/doi/10.1145/2903150.2911707`.

[10] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, editors. *Compilers: principles, techniques, & tools*. Pearson Addison-Wesley, Boston Munich, 2. ed., pearson internat. ed edition, 2007. ISBN 978-0-321-48681-3 978-0-321-49169-5.

[11] A. Aldini, R. Gorrieri, and F. Martinelli, editors. *Foundations of Security Analysis and Design III: FOSAD 2004/2005 Tutorial Lectures*. Number 3655 in Lecture Notes in Computer Science. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005. ISBN 978-3-540-28955-5 978-3-540-31936-8. doi: 10.1007/11554578.

[12] F. E. Allen. Control Flow Analysis. *SIGPLAN Notices*, 1970.

[13] R. Anderson, E. Biham, and L. Knudsen. Serpent: A Proposal for the Advanced Encryption Standard.

[14] A. W. Appel and J. Palsberg. *Modern Compiler Implementation in Java, Second Edition*.

[15] A. Bar-On. A 270 Attack on the Full MISTY.

[16] A. G. Bayrak, F. Regazzoni, D. Novo, P. Brisk, F.-X. Standaert, and P. Ienne. Automatic Application of Power Analysis Countermeasures. *IEEE Transactions on Computers*, 64(2):329–341, Feb. 2015. ISSN 0018-9340, 1557-9956, 2326-3814. doi: 10.1109/TC.2013.219. URL `https://ieeexplore.ieee.org/document/6671593/`.

[17] R. Beaulieu, D. Shors, J. Smith, S. Treatman-Clark, B. Weeks, and L. Wingers. The SIMON and SPECK Families of Lightweight Block Ciphers, 2013. URL `https://eprint.iacr.org/2013/404`. Publication info: Published elsewhere. Unknown where it was published.

[18] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. An efficient method of computing static single assignment form. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '89*, pages 25–35, Austin, Texas, United States, 1989. ACM Press. ISBN 978-0-89791-

294-5. doi: 10.1145/75277.75280. URL http://portal.acm.org/citation.cfm?doid=75277.75280.

[19] P. Dusart, G. Letourneux, and O. Vivolo. Differential Fault Analysis on A.E.S. In G. Goos, J. Hartmanis, J. Van Leeuwen, J. Zhou, M. Yung, and Y. Han, editors, *Applied Cryptography and Network Security*, volume 2846, pages 293–306. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003. ISBN 978-3-540-20208-0 978-3-540-45203-4. doi: 10.1007/978-3-540-45203-4_23. URL http://link.springer.com/10.1007/978-3-540-45203-4_23. Series Title: Lecture Notes in Computer Science.

[20] H. Eldib, C. Wang, and P. Schaumont. Formal Verification of Software Countermeasures against Side-Channel Attacks. *ACM Transactions on Software Engineering and Methodology*, 24(2):1–24, Dec. 2014. ISSN 1049-331X, 1557-7392. doi: 10.1145/2685616. URL https://dl.acm.org/doi/10.1145/2685616.

[21] A. Eubanks. The New Pass Manager, Mar. 2021. URL https://blog.llvm.org/posts/2021-03-26-the-new-pass-manager/. Section: posts.

[22] H. Gamaarachchi and H. Ganegoda. Power Analysis Based Side Channel Attack, Jan. 2018. URL http://arxiv.org/abs/1801.00932. arXiv:1801.00932 [cs].

[23] H. Gupta, S. Mondal, R. Majumdar, N. S. Ghosh, S. Suvra Khan, N. E. Kwanyu, and V. P. Mishra. Impact of Side Channel Attack in Information Security. In *2019 International Conference on Computational Intelligence and Knowledge Economy (ICCIKE)*, pages 291–295, Dubai, United Arab Emirates, Dec. 2019. IEEE. ISBN 978-1-72813-778-0. doi: 10.1109/ICCIKE47802.2019.9004435. URL https://ieeexplore.ieee.org/document/9004435/.

[24] D. Hong, J. Sung, S. Hong, J. Lim, S. Lee, B.-S. Koo, C. Lee, D. Chang, J. Lee, K. Jeong, H. Kim, J. Kim, and S. Chee. HIGHT: A New Block Cipher Suitable for Low-Resource Device. In D. Hutchison, T. Kanade, J. Kittler, J. M. Kleinberg, F. Mattern, J. C. Mitchell, M. Naor, O. Nierstrasz, C. Pandu Rangan, B. Steffen, M. Sudan, D. Terzopoulos, D. Tygar, M. Y. Vardi, G. Weikum, L. Goubin, and M. Matsui, editors, *Cryptographic Hardware and Embedded Systems - CHES 2006*, volume 4249, pages 46–59. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006. ISBN 978-3-540-46559-1 978-3-540-46561-4. doi: 10.1007/11894063_4. URL http://link.springer.com/10.1007/11894063_4. Series Title: Lecture Notes in Computer Science.

[25] D. Karaklajic, J.-M. Schmidt, and I. Verbauwhede. Hardware Designer's Guide to Fault Attacks. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*,

21(12):2295–2306, Dec. 2013. ISSN 1063-8210, 1557-9999. doi: 10.1109/TVLSI.2012.2231707. URL http://ieeexplore.ieee.org/document/6425517/.

[26] P. Kocher. Differential Power Analysis. 1999.

[27] P. Kocher, J. Jaffe, B. Jun, and P. Rohatgi. Introduction to differential power analysis. *Journal of Cryptographic Engineering*, 1(1):5–27, Apr. 2011. ISSN 2190-8508, 2190-8516. doi: 10.1007/s13389-011-0006-y. URL http://link.springer.com/10.1007/s13389-011-0006-y.

[28] 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. ISBN 978-0-7695-2102-2. doi: 10.1109/CGO.2004.1281665. URL http://ieeexplore.ieee.org/document/1281665/.

[29] P. Luo, K. Athanasiou, L. Zhang, Z. H. Jiang, Y. Fei, A. A. Ding, and T. Wahl. Compiler-Assisted Threshold Implementation against Power Analysis Attacks. In *2017 IEEE International Conference on Computer Design (ICCD)*, pages 541–544, Boston, MA, Nov. 2017. IEEE. ISBN 978-1-5386-2254-4. doi: 10.1109/ICCD.2017.94. URL http://ieeexplore.ieee.org/document/8119267/.

[30] S. Mangard, E. Oswald, and T. Popp. *Power analysis attacks: revealing the secrets of smart cards*. Springer, New York, 2007. ISBN 978-0-387-30857-9. OCLC: ocm71541637.

[31] M. Matsui and H. Ohta. A Description of the MISTY1 Encryption Algorithm. Request for Comments RFC 2994, Internet Engineering Task Force, Nov. 2000. URL https://datatracker.ietf.org/doc/rfc2994. Num Pages: 10.

[32] T. Mogensen. *Introduction to Compiler Design*. Undergraduate Topics in Computer Science. Springer International Publishing, Cham, 2024. ISBN 978-3-031-46459-1 978-3-031-46460-7. doi: 10.1007/978-3-031-46460-7. URL https://link.springer.com/10.1007/978-3-031-46460-7.

[33] National Institute of Standards and Technology (NIST). Data Encryption Standard. Jan. 1977.

[34] National Institute of Standards and Technology (NIST), M. J. Dworkin, E. Barker, J. R. Nechvatal, J. Foti, L. E. Bassham, E. Roback, and J. F. D. Jr. Advanced Encryption Standard (AES). *NIST*, Nov. 2001. URL https://www.nist.gov/publications/advanced-encryption-standard-aes. 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.

[35] F. A. P. Petitcolas. Kerckhoffs' Principle. In S. Jajodia, P. Samarati, and M. Yung, editors, *Encyclopedia of Cryptography, Security and Privacy*, pages 1–2. Springer Berlin Heidelberg, Berlin, Heidelberg, 2023. ISBN 978-3-642-27739-9. doi: 10.1007/978-3-642-27739-9_487-2. URL `https://link.springer.com/10.1007/978-3-642-27739-9_487-2`.

[36] S. Sanfilippo. *A compiler-based technique for automated analysis and protection against side-channel attacks.* PhD thesis, Politecnico di Milano, 2016.

[37] B. Schneier. CAST. In *Applied cryptography : protocols, algorithms, and source code in C*, pages 460–461. New York : Wiley, 1996. ISBN 978-0-471-12845-8 978-0-471-11709-4. URL `http://archive.org/details/Applied_Cryptography_2nd_ed._B._Schneier`.

[38] R. Sharp. *Introduction to Cybersecurity: A Multidisciplinary Challenge*. Undergraduate Topics in Computer Science. Springer Nature Switzerland, Cham, 2024. ISBN 978-3-031-41462-6 978-3-031-41463-3. doi: 10.1007/978-3-031-41463-3. URL `https://link.springer.com/10.1007/978-3-031-41463-3`.

[39] T. Shirai, K. Shibutani, T. Akishita, S. Moriai, and T. Iwata. The 128-Bit Blockcipher CLEFIA (Extended Abstract). In A. Biryukov, editor, *Fast Software Encryption*, pages 181–195, Berlin, Heidelberg, 2007. Springer. ISBN 978-3-540-74619-5. doi: 10.1007/978-3-540-74619-5_12.

[40] R. Spreitzer, V. Moonsamy, T. Korak, and S. Mangard. Systematic Classification of Side-Channel Attacks: A Case Study for Mobile Devices. *IEEE Communications Surveys & Tutorials*, 20(1):465–488, 2018. ISSN 1553-877X. doi: 10.1109/COMST.2017.2779824. URL `http://ieeexplore.ieee.org/document/8141882/`.

[41] STMicroelectronics. STM32F411xC STM32F411xE Datasheet - production data, Jan. 2024.

[42] D. J. Wheeler and R. M. Needham. TEA, a tiny encryption algorithm. In G. Goos, J. Hartmanis, J. Leeuwen, and B. Preneel, editors, *Fast Software Encryption*, volume 1008, pages 363–366. Springer Berlin Heidelberg, Berlin, Heidelberg, 1995. ISBN 978-3-540-60590-4 978-3-540-47809-6. doi: 10.1007/3-540-60590-8_29. URL `http://link.springer.com/10.1007/3-540-60590-8_29`. Series Title: Lecture Notes in Computer Science.

[43] William E. Burr and National Institute of Standards and Technology (NIST). Data Encryption Standard. In *A Century of Excellence in Measurements, Standards, and Technology*, pages 250–253. Jan. 2001.

[44] J. Yoon, S. Lee, d. h. cheon, J. Lee, and H. Lee. The SEED Encryption Algorithm. Request for Comments RFC 4269, Internet Engineering Task Force, Dec. 2005. URL `https://datatracker.ietf.org/doc/rfc4269`. Num Pages: 16.

List of Figures

1.1 General representation of a side channel attack 5
1.2 Circuit diagram of a CMOS inverter 9
1.3 Dynamic power consumption of a CMOS inverter [30] 10
1.4 Two power traces and their difference [27] 12
1.5 Differential power analysis attack scheme 14
1.6 The architecture of the MEET approach [8]. 19
1.7 Organization of the LLVM passes implemented and the visualization of the decency between them [36] 20

2.1 High level representation of a modular compiler 28
2.2 Detailed view of the phases of a compiler [10]. 29
2.3 Concrete syntax tree derived using the context-free grammar 2.1 31
2.4 Two examples of CFG in which is possible to observe to different properties of them 34
2.5 In the figure 2.5a there is a CFG and in the figure 2.5b there is its correspondent dominator tree. 37
2.6 A control flow graph from [14]. In gray the nodes of the dominance frontier of the node 5. 37
2.7 On the left there is a pseudocode with some assignment, and on the right the same pseudocode in SSA form 39
2.8 Example of data flow analysis showing the control flow graph and for each node the definitions the uses and the results of the liveness analysis and the reaching definitions. The variables c and m are parameters that are passed to this function from external. 43
2.9 Simplified scheme of the stages and the IRs used during a compilation of a C file using Clang 44
2.10 A simplified diagram of the LLVM IR layout. 46
2.11 Hierarchy of some of the most important LLVM APIs classes 51

3.1 Fragment of the CFG of the toy-cipher 59

3.2 Organization of the passes composing the tool 61
3.3 Identification phase 63

4.1 STM32F411E-DISCO board that was used. 77
4.2 Plot of hamming distances computed for the first three rounds of AES128. 84
4.3 Plot of hamming distances computed for the first three rounds of AES192. 84
4.4 Plot of hamming distances computed for the first three rounds of AES256. 85
4.5 Plot of hamming distances computed for the first three rounds of DES. . . 85
4.6 Plot of hamming distances computed for the first three rounds of HIGHT. 86
4.7 Plot of hamming distances computed for the first three rounds of MISTY1. 86
4.8 Plot of hamming distances computed for the first three rounds of SEED. . 87
4.9 Plot of hamming distances computed for the first nine rounds of SIMON128. 87
4.10 Plot of hamming distances computed for the first nine rounds of SIMON192. 88
4.11 Plot of hamming distances computed for the first nine rounds of SIMON256. 88
4.12 Plot of hamming distances computed for the first three rounds of SPECK128. 89
4.13 Plot of hamming distances computed for the first three rounds of SPECK192. 89
4.14 Plot of hamming distances computed for the first three rounds of SPECK256. 90
4.15 Plot of hamming distances computed for the sixteen three rounds of XTEA. 90

List of Tables

1.1 Power consumption simulation. 13
1.2 AES S-Box Table 13

2.1 Symbol table 28

4.1 Main characteristics of the analyzed ciphers. 77
4.2 All the possible configurations of Speck and Simon ciphers. 81
4.3 Parameters used to test each cipher, and number of Hamming distance functions generated. 83

List of Algorithms

2.1 Dominance frontier computation 36
3.1 Algorithm to find the type of a pointer 58
3.2 Use-Def Chains Identification 64
3.3 Key Schedule Identification 67
3.4 Register Writes Identification 70
3.5 MIR to IR Instructions Mapping 72
3.6 Unused Functions Removal 74

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 you 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.