

Formal methods in differential and linear trail search

Lieu du Projet de Fin d'Études

STmicroelectronics

Tuteur du Projet de Fin d'Études

Gilles VAN ASSCHE, Joan DAEMEN, Michael PEETERS

Correspondant pédagogique INSA Rennes

Barbara KORDY

PFE soutenu le 20/06/2016



life.augmented

Contents

1. Introduction	1
1.1. STMicroelectronics	1
1.2. Hash functions	2
1.3. Differential Cryptanalysis	2
2. Keccak	4
2.1. Sponge Construction	4
2.2. KECCAK- f	5
2.3. Differential Trails	7
2.3.1. Weight of a Differential Trail	7
2.3.2. Scan the Space and State Decomposition	8
2.3.3. Iterator and Pruning	9
2.4. Conclusion	9
3. A Semantic Iterator over a Semantic Tree	11
3.1. Coq and GALLINA	11
3.2. Definitions: Tree and Path	13
3.3. Specifications: Iterator	14
3.4. Proofs of the Tree Traversal Completeness	17
3.5. Pruning and Conclusion	18
4. From the Code to the Proofs	20
4.1. Hoare Logic	20
4.2. Proving the Soundness of the Iterators	22

5. When Proofs Provide Code	24
5.1. Which Language to Choose?	24
5.2. The Proof of Implication	25
5.3. The C++ Equivalence	27
5.4. The Chain of Trust and Conclusion	30
6. Conclusion	31
Bibliography	33
Global Internship Development	34
Pieces of code	36

Preface

While cryptography aims to provide different aspects of information security such as confidentiality, data integrity, entity authentication and data origin authentication [1], cryptanalysis aims to analyse, prove correct or if possible defeat, these techniques. For KECCAK, an algorithm that can be used for hashing, rather than trying to break it, its creators provided a measure (named *weight*) that gives an indication of its security against certain attacks. The higher the weight, the more resistant to cryptanalysis the algorithm will be. However, in order to find a lower bound for this weight, a computer-assisted search must be completed. Thus, the trust on the bound relies on the trust that the search is complete. The search algorithm has several iterators, each with their own “specialization” and each of them is written with so many considerable details, so there are many places where things can go wrong. The objective of this project is to explore how formal approaches can ensure that the search is complete, so that the lower bound for the weight is correct.

In formal methods, there are two approaches: either we have a specification and have already developed some software and we want to show that the software respects them eq. (1). This has been the approach of Appel et al for SHA-256 [2]. The alternative is to define a specification and try to extract a software from it which would then be proven by construction eq. (2).

$$\textit{Specification} \wedge \textit{Software} \rightarrow \textit{Prove the correctness of the software.} \quad (1)$$

$$\textit{Specification} \rightarrow \textit{Software} \quad (2)$$

In most cases, the second method is the one used, because it is easier to start from formal specification and proceed incrementally, through proven steps, to get to the software eventually.

This report is divided into five main parts.

- The first one (chapter 1) provides the notions required to understand the context of this internship.
- The second one (chapter 2) present the KECCAK algorithm and how differential cryptanalysis can be applied to it.
- In chapter 3, We provide a formal specification of an iterator and show its soundness with respect to the tree traversal.
- In chapters 4 and 5 I explore two approaches of formal methods (eq. (1) and eq. (2)). The first one shows how the current code can be verified. The second approach provides a generic implementation of an iterator of which the correctness is proven, given that the instantiation respects some provided properties.
- Finally in chapter 6, I provide a summary of this report. It shows what I learned and gives my thoughts about this internship.
- In the appendices, the reader will find the list of references, the Global Internship Development, and pieces of codes.

Related Work

The work of this internship aims to explore how formal approaches can be applied to verify the iterators used in *Differential propagation analysis of Keccak* by G. Van Assche and J. Daemen [3]. It is done in parallel with the work of Silvia Mella performing her PhD on the trail bounds of KECCAK.

Acknowledgments

This internship would not have been possible without the support of many people. Many thanks to my tutors Gilles VAN ASSCHE, Joan DAEMEN and Michael PEETERS, who answered my numerous questions and helped me make some sense through the Diffusion and Confusion. They provided me a great support and many different points of view which helped me tackle the problems I faced.

Many thanks to my long time friend Jean-Marie MADIOT who helped me when I had problems with Coq and gave me deep insights on my proofs. Thanks to the Coq-club who gave me a great deal of information about the way to establish a model of trees in Coq. I would also like to thanks Maxime VAN ASSCHE, who advised me about the possibility of this internship.

Many thanks to my parents and my engineering school, which provided financial support and without whom this internship would have never been possible.

And finally, I would like to thanks again my tutors for letting me spend some time answering questions on [crypto stackexchange](#) in order to sharpen my skills in cryptography.

Mathematical Background

$\text{GF}(2)$ also known as $\langle \mathbb{Z}_2, +, \times \rangle$ is the *Galois Field* (GF) of two elements: 0 and 1.

Because $\text{GF}(2)$ is a field,

- addition has an identity element (0) and an inverse for every element;
- multiplication has an identity element (1) and an inverse for every element but 0;
- addition and multiplication are commutative and associative;
- multiplication is distributive over addition.

One of the most interesting properties of $\text{GF}(2)$ is its link with the logical functions over booleans. An analogy can therefore be done between the two:

- The field addition (+) corresponds to the logical XOR operation (\oplus), also $\forall x \in \text{GF}(2), x + x = 0$.
- The field multiplication (\times) corresponds to the logical AND operation (\wedge).

Remarks: $0 + 1 = 1$ ($0 \oplus 1 = 1$) and $1 + 1 = 0$ ($1 \oplus 1 = 0$), therefore $x \mapsto x + 1$ is equivalent to the logical NOT. With this in mind, every booleans operations have an equivalent in $\text{GF}(2)$.

$\text{GF}(2)^n = \underbrace{\text{GF}(2) \times \cdots \times \text{GF}(2)}_{n \text{ times}}$ where \times is the direct product.

Acronyms

AES	Advanced Encryption Standard
CAESAR	Competition for Authenticated Encryption: Security, Applicability, and Robustness
CPA	Chosen Plaintext Attack
CPU	Central Processing Unit
D&I	Development & Innovation
DP	Differential Probability
GF	Galois Field
INSA	Institut National de Sciences Appliquées
ISS	Integrated Secure Solutions
MDG	Microcontrollers and Digital ICs Group
ML	MetaLanguage
MSM	Mechanized Semantics
MRI	Research Master's degree in Computer Science
POC	Proof of Concept
QSC	Quality, Support and Control
R&D	Research and Development
S-box	substitution-box
SGS	Società Generale Semiconduttori
SHAKE	Secure Hash Algorithm <small>KECCAK</small>
SHA-3	Secure Hash Algorithm 3
SMD	Secure Microcontroller Division
STM	STMicroelectronics
VST	Verified Software Toolchain
XOF	eXtendable Output Functions

1. Introduction

In this chapter, I provide a brief introduction of the company in which this internship has been completed. Then a presentation of hash functions and the differential cryptanalysis is given in order to provide a wide vision of the context of the subject.

1.1 STMicroelectronics



Figure 1.1: STMicroelectronics logo

STMicroelectronics (**STM**) is a global semiconductor company with net revenues of US\$ 6.90 billion in 2015. Offering one of the industry's broadest product portfolios, **STM** serves customers across the spectrum of electronics applications with innovative semiconductor solutions for Smart Driving and the Internet of Things.

	Company	Revenue (million \$USD)	Market Share
1	Intel Corporation	49 964	14.1%
2	Samsung Electronics	38 273	10.8%
3	Qualcomm	19 266	5.5%
4	Micron Technology	16 389	4.6%
5	SK Hynix	15 737	4.5%
6	Texas Instruments	11 420	3.5%
7	Toshiba Semiconductor	8 496	2.4%
8	Broadcom	8 387	2.4%
9	STMicroelectronics	7 395	2.1%
10	MediaTek	7 194	2.0%

Table 1.1: STMicroelectronics in the semiconductor market in 2014

STM was created in 1987 by the merger of two long-established semiconductor companies, *Società Generale Semiconduttori* (**SGS**) Microelettronica and Thomsom Semiconducteurs of France, and has been publicly traded since 1994. Its shares trade on the New York Stock Exchange, on Euronext Paris and on Borsa Italiana.

The group has approximately 43200 employees; 8300 people working on *Research and Development* (**R&D**); 11 manufacturing sites; advanced research and development centres in 10 countries; and sales offices all around the world. To keep its technology edge, **STM** has maintained an unwavering commitment to **R&D** since its creation. In 2015 the Company spent about 21% of its revenue on **R&D**. Among the industry's

most innovative companies, **STM** owns and continuously refreshes a substantial patent library (15,000 owned patents in 9,000 patent families and 500 new patent filings).

In Belgium

STM has a software-dedicated division in Brussels (Diegem). It is actually a department of *Integrated Secure Solutions* (ISS) which is part of the *Secure Microcontroller Division* (**SMD**), part of *Microcontrollers and Digital ICs Group* (**MDG**) of **STM**.

The technical part of the division is divided into two departments: “*Development & Innovation* (**D&I**)” department and “*Quality, Support and Control* (**QSC**)” department.

Official ST name	ST Zaventem
Country	Belgium
City	Diegem
Address	STMicroelectronics, Green Square Building B, Lambroekstraat 5, B-1831, Diegem/Mechelen, Belgium
Phone	+32 2 724 5111
Site Activity	Design center, Sales office
Site Manager	Armand LINKENS
Employees	60 ~ 70

Table 1.2: STMicroelectronics Zaventem

Some members of the **D&I** department are also actively taking part in cryptographic research with the KEYAK team. They submitted KEYAK, a new mode of encryption combined with a sponge construction and the KECCAK- f permutation at the current *Competition for Authenticated Encryption: Security, Applicability, and Robustness* (**CAESAR**).

1.2 Hash functions

A hash function maps a bit string input of variable length to a bit string output of fixed length eq. (1.1):

$$H : \{0, 1\}^* \mapsto \{0, 1\}^n \quad (1.1)$$

Consider a hash function h with an output of size n . We define the following notions:

Collision resistance: it is difficult to find two input a and b such as $a \neq b$ and $h(a) = h(b)$.
The complexity of a generic attack is $\mathcal{O}(2^{n/2})$ (birthday paradox).

Preimage resistance: given a hash c , it is difficult to find b such as $h(b) = c$.

Second-preimage resistance: given an input a , it is difficult to find b such as $a \neq b$ and $h(a) = h(b)$.
The complexity of finding a (second) preimage is at most $\mathcal{O}(2^n)$ (brute force).

From a cryptographic point of view, a hash function is expected to satisfy the above properties and therefore must be difficult to invert. Some of the applications of cryptographic hash functions are the verification of the integrity of content and the secure storage of passwords used for authentication.

1.3 Differential Cryptanalysis

Cryptanalysis is the study of mathematical techniques for attempting to defeat cryptographic techniques, and, more generally, information security services.

Handbook of applied cryptography [1]

Differential Cryptanalysis has been introduced by Biham and Shamir at Crypto 90 [4]. In *Chosen Plaintext Attack (CPA)*, the idea is to exploit statistical information on differences between chosen plaintexts in order to retrieve the key in the case of a cipher or to find a second-preimage in the case of hash function.

Given two input t and t' , a difference Δ is the value of $(t \oplus t')$ where \oplus is the xor operation. A differential is composed of two differences and written as $(\Delta_1 \Rightarrow \Delta_2)$. The idea behind a differential is to show that given an input difference Δ_1 , chances are that a difference Δ_2 will occur (see Fig. 1.2). This chance is called the *Differential Probability (DP)*.

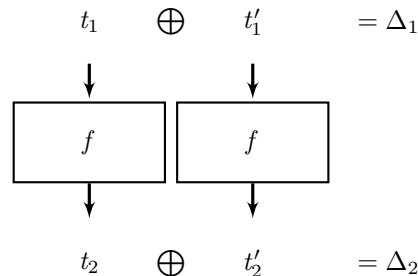


Figure 1.2: A differential $(\Delta_1 \Rightarrow \Delta_2)$

Applied to a function (usually called a round function) iterated a number of times, with multiple differentials, it is possible to build a trail e.g. $(\Delta_1 \Rightarrow \Delta_2 \Rightarrow \dots \Rightarrow \Delta_k)$. In the general case of a hash function, a collision trail is a trail where the first difference is non-null and the last difference is null. Using this trail one finds a collision by finding a pair of inputs that match for each round differences in the trail. The difficulty to find such pair is related to the differential probability of the trail.

Weight of a Trail

Another way to quantify the Differential Probability is to see it as a weight eq. (1.2).

$$DP = \frac{1}{2^w} \quad (1.2)$$

Another definition is provided by Daemen et al.[3]:

For a function f with domain $\text{GF}(2)^b$, the weight can also be seen as :

$$w(\Delta_1 \xRightarrow{f} \Delta_2) = b - \log_2 \#\{t \in \text{GF}(2)^b : f(t) + f(t + \Delta_1) = \Delta_2\} \quad (1.3)$$

If the argument of the logarithm is non-zero (i.e. the DP is non-zero), the differences Δ_1 and Δ_2 are said *compatible*. Otherwise, the weight is undefined.

The **weight of a trail is the sum of the weight of the differentials that compose this trail.**

The greater the weight, the harder it will be to exploit the differential. It is also noteworthy that **affine applications have no influence on the DP of differentials.**

Proof: Let

- $K \in \text{GF}(2)^n$ be a constant;
- A a matrix of $\text{GF}(2)^n$;
- $f : \text{GF}(2)^n \rightarrow \text{GF}(2)^n$ such as $f(x) = Ax + K$;
- Δ_1 and Δ_2 two differences

$$P(\Delta_1 \xRightarrow{f} \Delta_2) > 0 \Leftrightarrow \exists t, \Delta_2 = f(t) + f(t + \Delta_1). \quad (1.4)$$

$$\Leftrightarrow \exists t, \Delta_2 = At + K + A(t + \Delta_1) + K. \quad (1.5)$$

$$\Leftrightarrow \Delta_2 = A\Delta_1. \quad (1.6)$$

Therefore the DP of a difference over an affine application is 1. □

2. Keccak

KECCAK, the winning algorithm of the **SHA-3** competition is the combination of a sponge construction with an invertible permutation named KECCAK- f . In this chapter we will briefly explain the sponge construction before moving to the KECCAK- f permutation. Then we will present the strategy used to prove the bounds of the differential trails.

2.1 Sponge Construction

The sponge construction [5] is used to build an *eXtendable Output Functions* (XOF). It takes a variable-length input and returns an output of length n eq. (2.1).

$$F : \text{GF}(2)^* \rightarrow \text{GF}(2)^n \quad (2.1)$$

Therefore it can be used as a hashing function or a stream cipher. Such a function uses a state of $b = r + c$ bits. b is called the width, r the bitrate and c is the capacity. For a security claim S , the complexity of any attacks is the minimum between $2^{S/2}$ and the complexity of the same attack on a *random oracle*¹. In **SHA-3** and **SHAKE** functions, the security claim is $c/2$. Hence for **SHA-3-256** (as 256 bits of security claim and 256 bits of output) and for **SHAKE256** (as 256 bits of security claim, extendable output) we have $c = 512$ and $r = 1088$. This construction is divided into two phases.

1. Absorption phase: input is divided into blocks of r bits and XORed into the state.
2. Squeezing phase: the first r bits are extracted from the state.

Between each absorption or squeezing, the transformation function f is called in order to randomize the state. Note that the last c bits are never affected by the input, nor displayed in the output. They are only modified by the transformation function f .

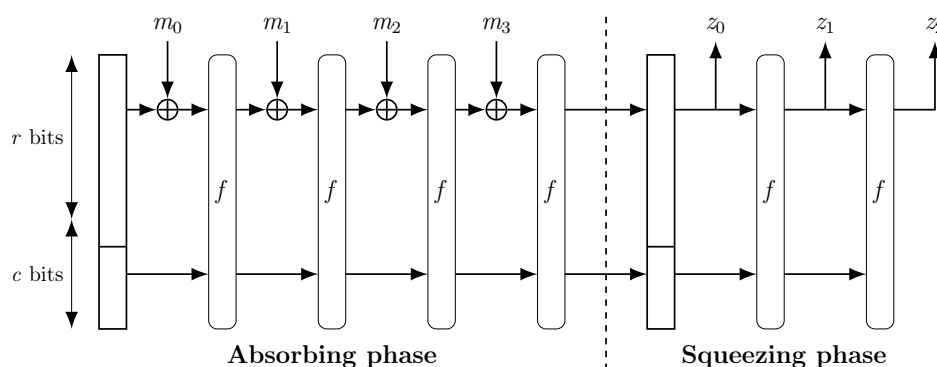


Figure 2.1: A sponge construction

¹<http://crypto.stackexchange.com/a/22357/29161>

2.2 KECCAK- f

KECCAK- f is a family of seven permutations denoted by KECCAK- $f[b]$ with a width $b \in \{25, 50, 100, 200, 400, 800, 1600\}$. The state of KECCAK- f is organized as a parallelepiped of dimension $5 \times 5 \times w$ where $w \in \{1, 2, 4, 8, 16, 32, 64\}$. Each bit can be positioned by its coordinates (x, y, z) (See Fig. 2.2). A bit is *active* if it has value 1 and *passive* otherwise. For a better visualisation of the active bit position inside the state, a lighter representation will equally be used (see Fig. 2.3).

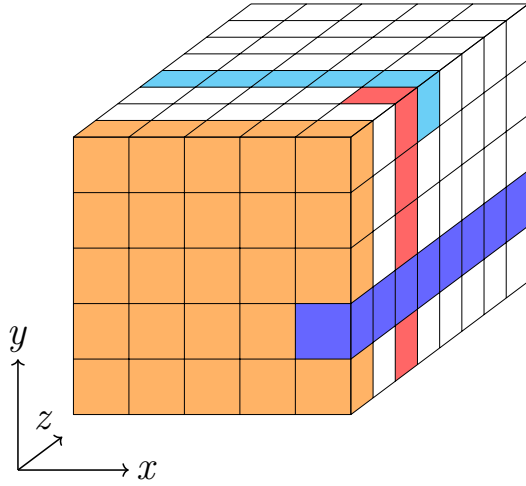


Figure 2.2: State structure of KECCAK- $f[200]$

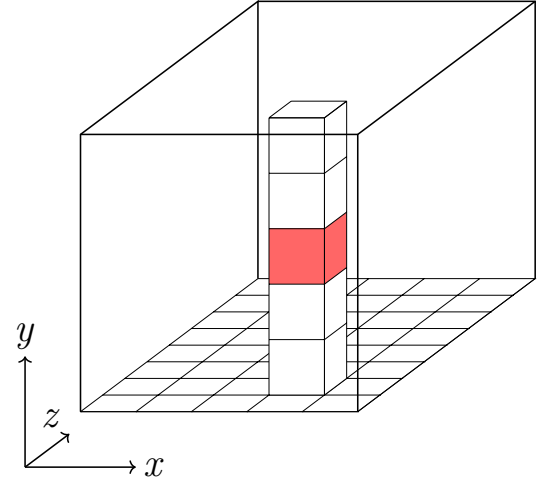


Figure 2.3: Light representation of the state structure where the bit $(3, 2, 1)$ is active.

Given this representation of a state, the following notions are defined:

- a slice is a set of 25 bits with fixed z coordinate (orange in Fig. 2.2).
- a lane is a set of w bits with fixed (x, y) coordinate (blue in Fig. 2.2).
- a row is a set of 5 bits with fixed (y, z) coordinates (cyan in Fig. 2.2).
- a column is a set of 5 bits with fixed (x, z) coordinates (red in Fig. 2.2).

KECCAK- f is composed of $12 + 2 \times \log_2(w)$ iterations of a round permutation composed of 5 transformations: $f = \iota \circ \chi \circ \pi \circ \rho \circ \theta$.

θ (theta)

θ is a linear mixing layer which operates on columns.

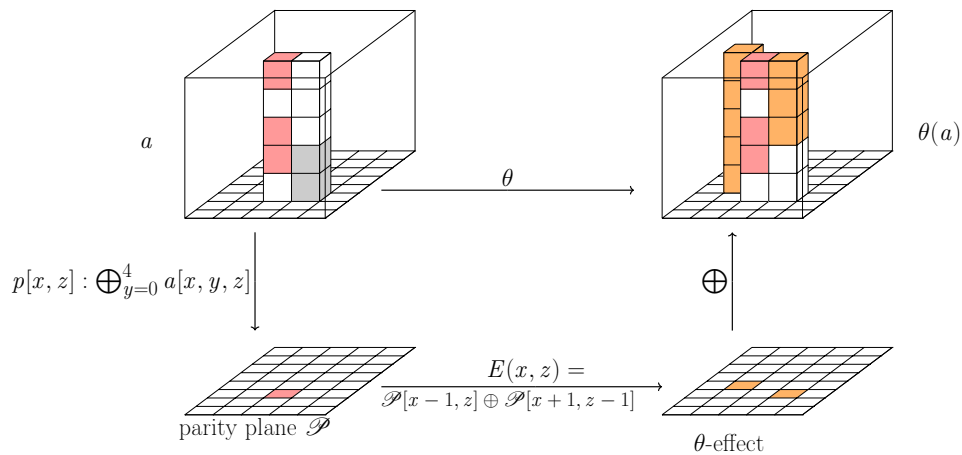


Figure 2.4: How θ is applied to a state. Active bits are coloured (red, orange, grey).

The parity function (p) computes the parity plane \mathcal{P} as the sum over the columns. From this plane the θ -effect can be deducted before being added back to the state. This parity will be an important element later in the search of the bounds.

ρ (rho) and π (pi)

Both ρ and π operate on lanes. While ρ is a bit-wise cyclic shift (Fig. 2.5), π transposes the position of the lanes (Fig. 2.6). Because θ operates on columns, it is important to separate every bit of a column after one application to avoid the appearance of patterns. Therefore π guarantees that all bits in a column are evenly spread over the slice. The role of ρ is make sure that differences are spread over slices. These two transformations aim to propagate the modifications applied to the state.

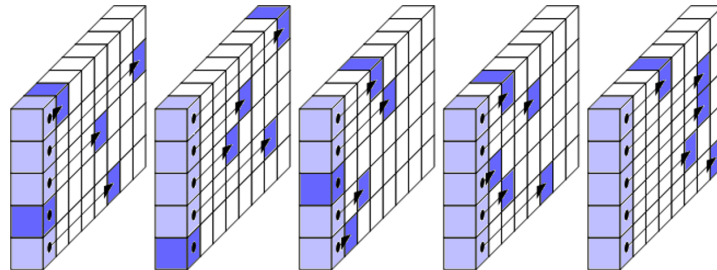


Figure 2.5: The ρ transformation

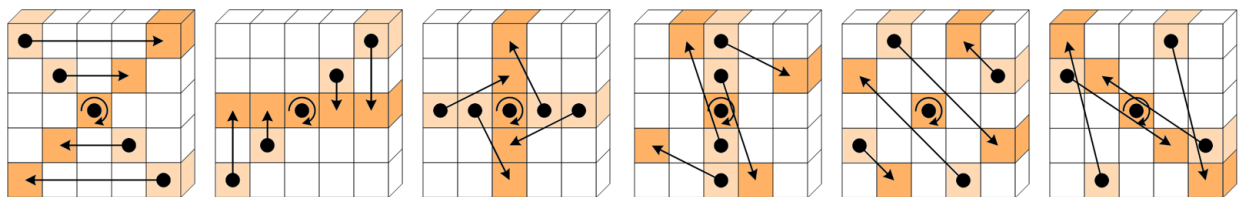


Figure 2.6: The π transposition

χ (chi)

χ is the only non-linear mapping in $\text{KECCAK-}f$. Without it, the round function of $\text{KECCAK-}f$ would be linear [6] (which would imply a big weakness against differential cryptanalysis). χ has an algebraic degree of 2 and operates on rows. Each application of χ could be seen as the application of a 5-bit *substitution-box* (*S-box*).

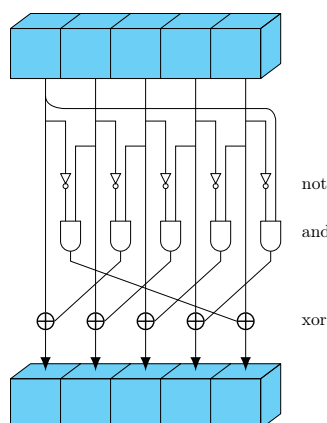


Figure 2.7: The χ transformation

ι (iota)

ι consists of the addition of round constants to the lane $(0,0)$ and is aimed at disrupting symmetry. Without it, the round function would be translation-invariant in the z direction and all rounds would be equal making KECCAK- f subject to attacks exploiting symmetry such as slide attacks [6].

2.3 Differential Trails

In this section, I present how the weight of the trails is calculated and the (simplified) strategy used to find the bounds.

2.3.1 Weight of a Differential Trail

In section 1.3, we defined the weight of a trail as the sum of the weights of the round differentials that compose this trail.

Let Q be a trail of differences $a_0, a_1, a_2, \dots, a_n$:

$$Q = a_0 \xrightarrow{\iota \circ \chi \circ \pi \circ \rho \circ \theta} a_1 \xrightarrow{\iota \circ \chi \circ \pi \circ \rho \circ \theta} a_2 \xrightarrow{\iota \circ \chi \circ \pi \circ \rho \circ \theta} \dots \xrightarrow{\iota \circ \chi \circ \pi \circ \rho \circ \theta} a_n \quad (2.2)$$

We can calculate its weight.

$$w(Q) = \sum_{i=0}^{n-1} w(a_i \xrightarrow{\iota \circ \chi \circ \pi \circ \rho \circ \theta} a_{i+1}) \quad (2.3)$$

This trail can also be decomposed as:

$$Q = a_0 \xrightarrow{\pi \circ \rho \circ \theta} b_0 \xrightarrow{\chi} c_0 \xrightarrow{\iota} a_1 \xrightarrow{\pi \circ \rho \circ \theta} b_1 \xrightarrow{\chi} c_1 \xrightarrow{\iota} a_2 \xrightarrow{\pi \circ \rho \circ \theta} \dots \xrightarrow{\chi} c_{n-1} \xrightarrow{\iota} a_n \quad (2.4)$$

In the previous section, we saw that KECCAK- f is composed of affine functions (θ, ρ, π and ι) and a non-linear one (χ). In section 1.3, we showed that affine functions have a DP of 1. Only the form of the difference changes. Thus the affine part does not increase the weight of the trail. Because ι is the addition of a constant, the difference c_{i-1} is equals to the difference a_i .

Therefore, the weight of a trail can be simplified as follows:

$$w(Q) = \sum_{i=0}^{n-1} w(b_i \xrightarrow{\chi} a_{i+1}) \quad (2.5)$$

For a given input difference b_n , the space of compatible output differences a_{n+1} through χ forms a linear affine variety $\mathcal{A}(b_n)$ [7, 6].

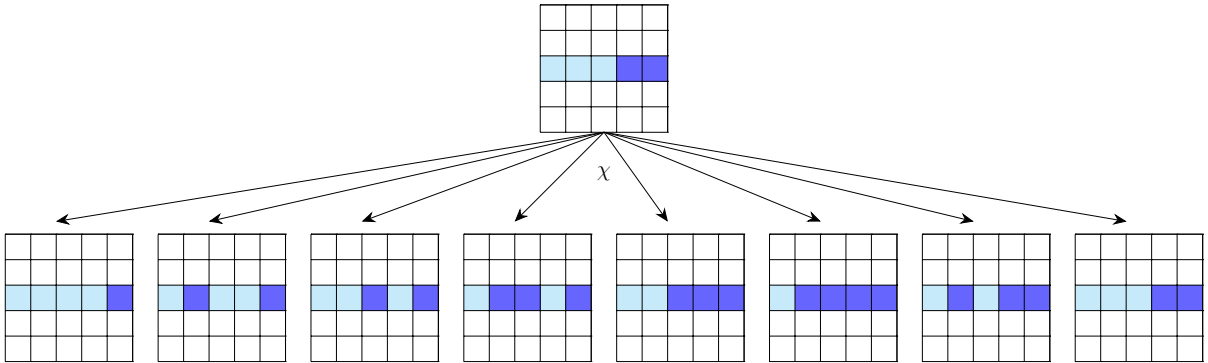






Figure 2.8: For a given input difference in a row, list of possible differences after χ . A dark blue square means an active bit.


Because χ has an invariance by translation over x , it is possible to summarize all possible differences. Table 2.1 displays the offset and bases for the affine spaces of all single-row differences. Each line provides the *offset* and the *base* to build all the output differences of a given input.

Example: For an input difference 00011  (as in Fig. 2.8), the offset is 00001, and the base (a_0, a_1, a_2) is (00010, 00100, 01000). The resulting affine space is the set of elements built from the linear combinations of this base: $offset + k_0 \cdot a_0 + k_1 \cdot a_1 + k_2 \cdot a_2 \in \mathcal{A}$

00001 + 0 · 00010 + 0 · 00100 + 0 · 01000 = 00001 

00001 + 0 · 00010 + 0 · 00100 + 1 · 01000 = 01001 

00001 + 0 · 00010 + 1 · 00100 + 0 · 01000 = 00101 

00001 + 0 · 00010 + 1 · 00100 + 1 · 01000 = 01101 

and so on...

input difference	propagation through χ					$w(.)$	$\ \cdot \ $
	offset	base elements					
		a_0	a_1	a_2	a_3		
00000	00000					0	0
00001	00001	00010	00100			2	1
00011	00001	00010	00100	01000		3	2
00101	00001	00010	01100	10000		3	2
10101	00001	00010	01100	10001		3	3
00111	00001	00010	00100	01000	10000	4	3
01111	00001	00011	00100	01000	10000	4	4
11111	00001	00011	00110	01100	11000	4	5

Table 2.1: Space of possible output differences, weight, and Hamming weight of all row differences, up to cyclic shifts [3].

From the table of possible outputs, we can extract interesting pieces of information:

- the weight $w(b_i \xrightarrow{\chi} a_{i+1})$ only depends on b_i and is equal to $w(b_i) \triangleq \log_2(\#\mathcal{A}(b_i))$,
- as χ operates on each row independently, the weight can also be computed on each row independently and then summed,
- and more importantly, **adding active bits to the state will never decrease the weight.**

2.3.2 Scan the Space and State Decomposition

In RIJNDAEL [8] (the winning algorithm of the *Advanced Encryption Standard (AES)* competition [9]), a difference propagation nicely follows the byte boundaries. It is said that the RIJNDAEL round function has *strong alignment* with respect to bytes [10]. The position of difference propagation is therefore predictable. In KECCAK, due to the linear part of the round function (θ, ρ and π), difference propagation is harder to predict. There is no structure such as the byte for RIJNDAEL that allows to restrict the position of a difference. This is called a *weak alignment*.

Because of this property in KECCAK, we must scan the space of the differences in order to find the minimum weight of a differential trail over KECCAK- f . The idea is therefore to prove the absence (or existence) of a n -round trail under specified weight (Daemen et Van Assche proved the absence of 3-round trails under a weight of 32 [3]).

Kernel, Parity, Orbitals and Runs

To simplify the search, it is possible to decompose a state pattern as an ordered list of simple elements. Using this definition we can build a tree where the parent of a node is the state pattern with the last element removed. The sibling of a node is the state pattern with the last element changed by its successor. For KECCAK- f , these units are called **runs** and **orbitals**. Using properties over θ , we can build the states from the parity plane (\mathcal{P}). We need to consider two cases:

- When the parity plane is empty. The corresponding subspace is called the kernel and is the mathematical kernel of the parity function defined in section 2.2. All columns in this subspace are called *even*.
- When the parity plane has active bits, the columns associated to these active bits are called *odd*.

In the first case, θ is equivalent to the identity. However in the second case, we will have to calculate the θ -effect which will flip some columns called *affected*. The positions of the *affected columns* on \mathcal{P} form pairs, associated with their related *odd columns*, they form a **run**. Given a state, we can add bits by pairs in the same columns as they do not change the parity. These pairs are called **orbitals**. A column has multiple orbital configurations possible on which we can iterate (see Fig. 2.9).

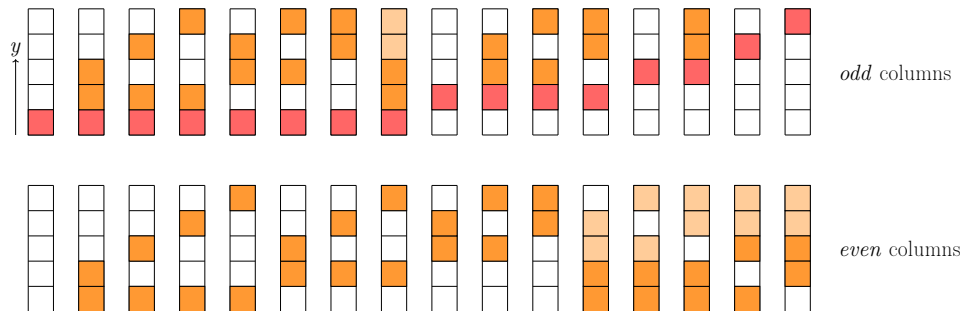


Figure 2.9: The possible configurations of bits inside a column

To build our state from the parity plane, we first iterate over the runs (in absence of runs we are in the kernel) and then we iterate over the orbitals. The full decomposition of the tree can be seen in Fig. 2.10.

2.3.3 Iterator and Pruning

As **adding active bits never decreases the weight of the differentials**, we can define a monotonic bounding function. Because we want to prove the absence (or existence) of a n -rounds trails under a certain weight W_{limit} . We can therefore build all the trails with a cost up to W_{limit} and stop adding *runs* or *orbitals* when it exceeds the budget, cutting the branches. This effectively prunes the tree and highly decreases the vast search space.

In order to iterate on the different kinds of elements: *orbitals*; *odd columns*; *affected columns*; *runs*, iterators have been defined. However they require to interact between each other in a complex way (in particular due to the pruning). Moreover we have to make sure that no possible state is skipped, in other words ensure the completeness of the iterator search.

2.4 Conclusion

In this chapter, we briefly described the KECCAK function. We tried to provide an adequately detailed explanation of the context of this internship and motivate the research we have been performing. The idea to prove the absence of n -rounds trails under a certain weight requires multiple usage of specialized and complex iterators in a tree traversal. The aim of my work was to prove the soundness and the completeness of their search.

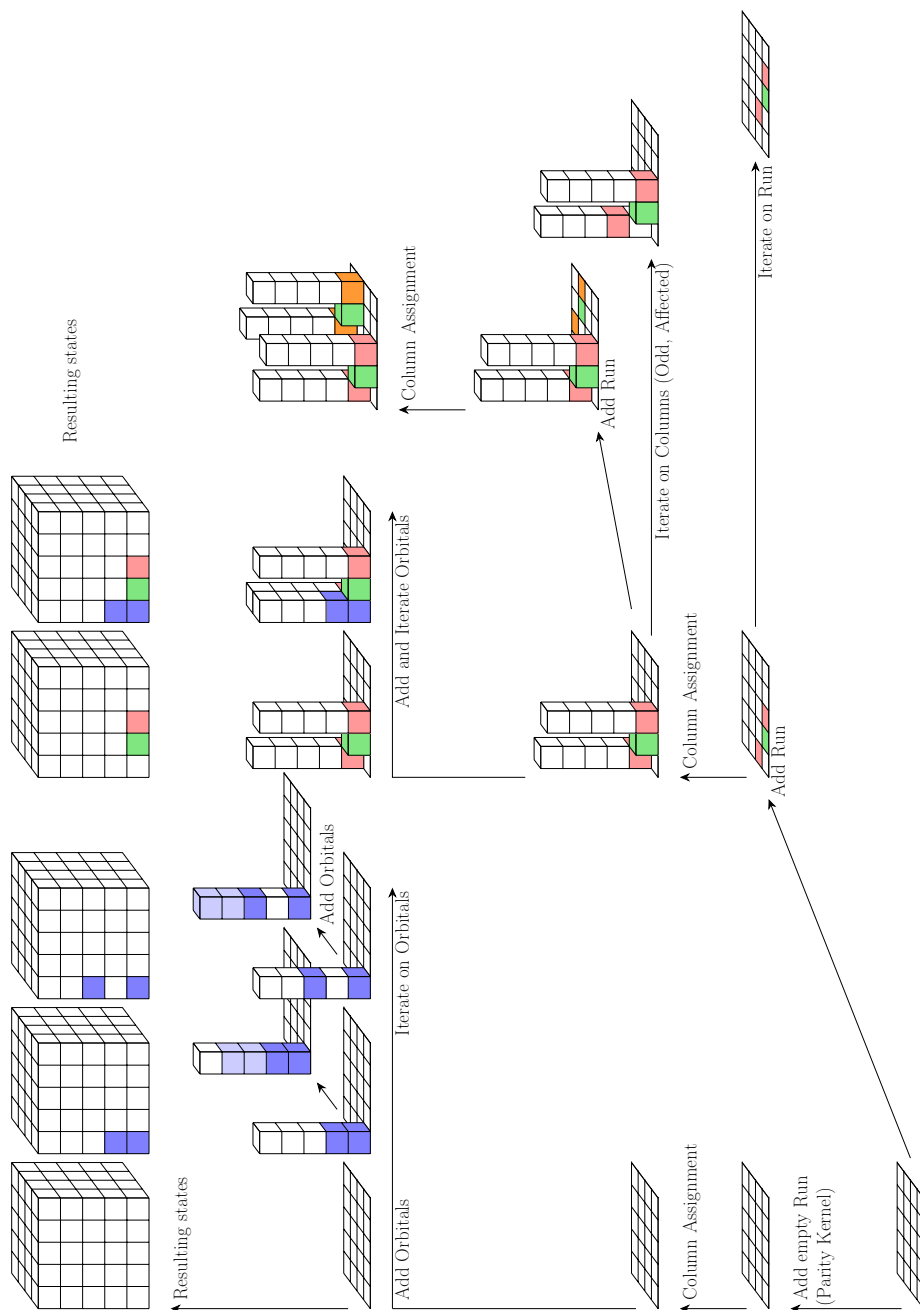


Figure 2.10: Tree decomposition of the search

3. A Semantic Iterator over a Semantic Tree

In order to be able to prove the soundness of the iterators, one must have a formal specification of the different kinds of elements we will be working on. Before giving the specification, I will provide a quick presentation of Coq. Then I will describe the representation of the tree and the iterator. Afterward, I will prove the soundness of the specification and tackle the pruning problem.

3.1 Coq and GALLINA

Coq [11] is a formal proof management system. It helps and ensures the correctness of the steps in a proof. Let us assume someone wants to prove " $\forall n \in \mathbb{N}, P(n)$ ". By reasoning by induction over n , Coq will ask the user to prove $P(0)$. On completion it will add $P(n)$ in the hypothesis and ask to prove $P(n + 1)$. It guarantees that no steps has been omitted in the proof. This provides a strong confidence but also forces the user to justify every step he uses.

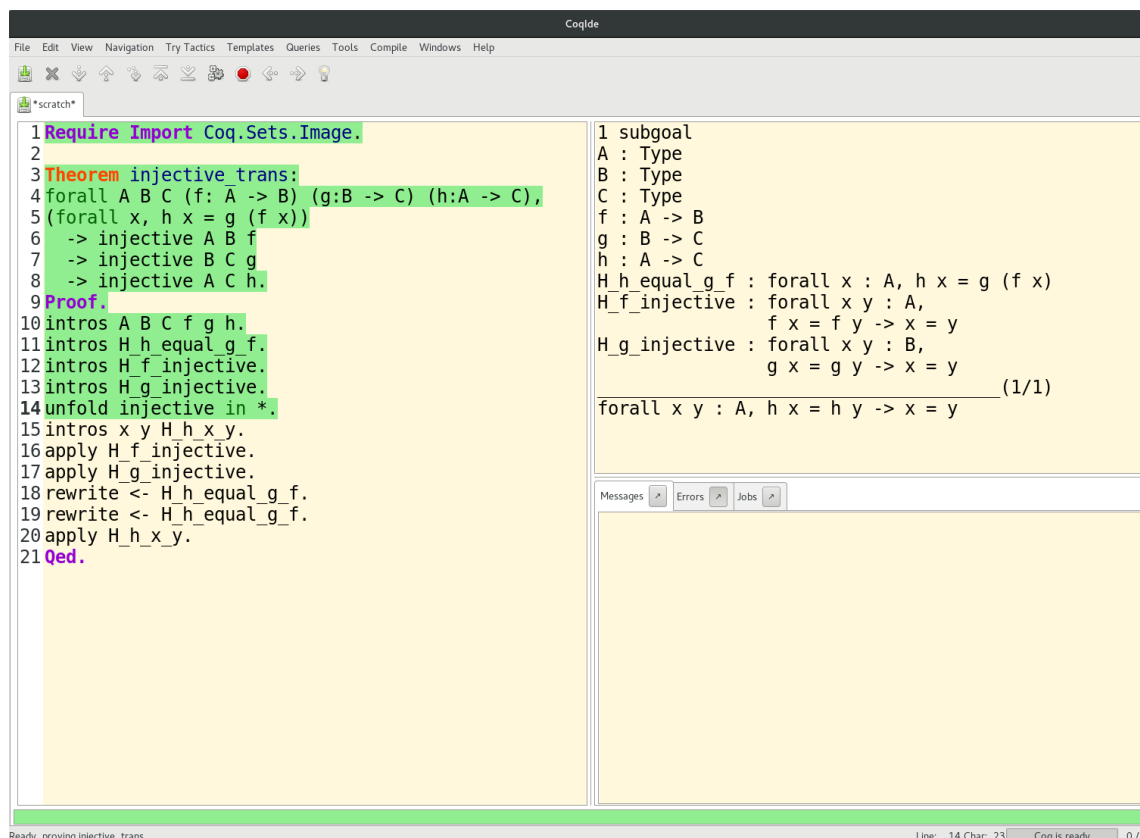


Figure 3.1: CoqIde in the proof of the transitivity of the injectivity through the composition. On the left lies the script of the proof, while on the right we see the current goal and the list of hypotheses.

Coq uses the GALLINA language, based on the functional paradigm. The code can also be exported to Ocaml and therefore natively compiled. This allows to work on the specifications with Coq and extract formally proven pieces of code. The code listing in Code 3.1 provides a quick introduction to the syntax used by GALLINA.

```
(* This is a comment *)

(* Definitions *)
Theorem ...      (* defines a theorem *)
Lemma ...        (* defines a lemma *)
Corollary ...    (* defines a corollary *)
Definition ...   (* defines a function *)
Fixpoint ...     (* defines a recursive function *)
Inductive ...    (* defines an inductive definition: useful for induction *)

(* Proving *)
Proof            (* start a proof *)
Qed              (* save a completed proof *)
Admitted         (* admit a proof (this is cheating !) *)

(* The types Proposition *)
Prop            (* as Proposition, this is the type used in a proof *)
False           (* something false *)
True            (* something true *)
^              (* logical and *)
V              (* logical or *)
∃              (* exist *)
∀              (* for any *)
→              (* implies *)
↔              (* if and only if *)
~              (* not, ~P is equivalent to P → False *)

(* The boolean type *)
false true      (* There is a separation between booleans and propositions. *)
andb            (* boolean version of and *)
orb            (* boolean version of or *)
negb           (* boolean version of not *)

(* About functions *)
(f: A → B)      (* a function which takes an input of type A and a return of type B. *)
(f: A → B → C)  (* idem with two inputs. One of type A, one of type B and return of type C. *)
f x y z         (* the function f is applied to x, y and z, equivalent to f(x,y,z) in C *)

(* case analysis *)
match l with    (* this construct a pattern matching, sort of equivalent to an if else elseif. *)
| ...           (* case 1 *)
| ...           (* case 2 *)
| ...           (* case 3 *)
end             (* Coq makes sure that no cases are missed. An error is raised if missing. *)

(* tactics *)
induction n     (* start an induction on the term n *)
apply H         (* apply the hypothesis/lemma/... named H on the goal *)

(* example of a recursive function that defines the property "element a is in the list l" *)
Fixpoint In (a:A) (l:list A): Prop:=
  match l with
  | [] ⇒ False (* list is empty → return False *)
  | b :: q ⇒ b = a ∨ In a q (* list has a head (b) and a queue q *)
  end.
```

Code 3.1: The GALLINA language

Remark: When using **Fixpoint**, Coq searches for a decreasing argument in order to prove the termination of the algorithm.

The Option Type

A useful `Type` in Coq is `option A` where `A` is another `Type`. It allows the definition of functions with a sort of *exception* output. A simple application of it is in the retrieval of the n^{th} element of a list (`nth_error`). If the list is too short, it returns `None` otherwise it returns `Some x` where `x` is the desired element.

```
Inductive option (A:Type) : Type :=
| Some : A → option A
| None : option A.

Theorem SomeEq: ∀ A (a b:A), Some a = Some b ↔ a = b.
```

Code 3.2: The option type

3.2 Definitions: Tree and Path

Using the `GALLINA` we can define the different kinds of elements on which we will work on.

Tree

Because the number of branches for a node varies, we will consider a node as a value (of type `x`) and a list of sub-trees (`list Tree`) (See Code 3.3). node can also be seen as a function with two arguments. It is called a *constructor*. Coq allows us later to reason on this `Inductive` structure by building the right hypothesis for the induction principle. Unfortunately this scheme is slightly more complex than the basic ones (such as `list X`) therefore it will not be able to generate the desired induction scheme (see section 3.8 in [12]). We need to define our induction principle¹[13].

```
Section trees.
  Variable (X : Type).

  (* we do not want the too weak Coq generated induction principles *)
  Unset Elimination Schemes.

  Inductive Tree : Type := node : X → list Tree → Tree.
  Set Elimination Schemes.

  Section Tree_ind.
    Variable P : Tree → Prop.
    Hypothesis HP : ∀ a ll, (∀ x, In x ll → P x) → P (node a ll).

    Definition Tree_ind : ∀ t, P t.
  End Tree_ind.
End trees.
```

Code 3.3: Tree definition

The induction principle can be illustrated as:

1. Prove the property for a tree with no children.
2. Assume that the property is `True` for all children, prove it for the parent.

Path

In order to access an element of the tree, we need a `Path`. It can be seen as a list of natural numbers (Code 3.4) which defines the index of the child in the list of the subtrees. For example, the empty list (`[]`) returns the root, the list `[0]` returns the first child (Code 3.5).

```
Definition Path := list nat.
```

Code 3.4: Path definition

¹Thanks to Dominique Larchey-Wendling for the clear explanations.

```

Fixpoint getNodeApp X (p:Path) (t:option (Tree X)) : option (Tree X) :=
  match t,p with
  | None, _ => None
  | v, [ ] => v
  | Some (node v l), (h :: q) => match nth_error l h with
    | None => None
    | t => getNodeApp X q t
  end
end.

Arguments getNodeApp [X] _ _ .

Definition getNode X (p:Path) (t:Tree X) : option (Tree X) :=
  getNodeApp (rev p) (Some t).

Arguments getNode [X] _ _ .

```

Code 3.5: Given a path we can return the selected subtree with getNode

Given that we defined a function to access nodes given a path, we can deduce properties (Code 3.6) and another induction principle (Code 3.7).

```

Lemma noDown : ∀ X h p t (v: X),
  getNode p t = Some (node v [ ] ) → getNode (h::p) t = None.
(* If a path p is applied to a tree t returns a node without subtree,
   extending this path will return nothing. *)

Lemma compoGetNode: ∀ X l1 l2 (t:Tree X) t2,
  getNode l1 t = Some t2 → getNode (l2 ++ l1) t = getNode l2 t2.
(* If a path l1 applied to a tree t returns a node t2,
   extending l1 with l2 have the same result as applying l2 on the t2. *)

```

Code 3.6: Simple properties on getNode

```

Section Tree_ind2.
  Variable X : Type.
  Variable P : Tree X → Prop.

  Theorem InGetNodEq : ∀ a l1 p tree,
    getNode p tree = Some (node a l1) →
    (∀ x, In x l1 → P x) ↔ (∀ n x, getNode (n::p) tree = Some x → P x).

  Hypothesis HP : ∀ tree,
    (∀ n x, getNode (n:: [ ] ) tree = Some x → P x)
    → P tree.

  Definition Getnode_ind : ∀ t, P t.
End Tree_ind2.

```

Code 3.7: Second induction principle over Trees

3.3 Specifications: Iterator

We defined the tree with a generic type (`Tree X`), we also defined a way to access a node in this tree by using a stack as a path. We will now consider the traversal of this tree.

Move

On a node there are several possible moves. The idea is to be able to define the next movement in function of the current node and the previous movement. Such as: if the last move was *to the parent*, the next move must not be *to the child*.

```

Inductive MoveSS : Type := TO_PARENT | TO_CHILD | TO_SIBLING | VISITED.

```

Code 3.8: Definition of the movements

In Code 3.8, we also defined the *movement* VISITED. While its usage might be seen as a duplicate of TO_PARENT, the idea here is to provide an intermediate step that defines that we have visited all the current subtree and we can move to the next one (either by a move TO_PARENT or TO_SIBLING) (see Fig. 3.2).

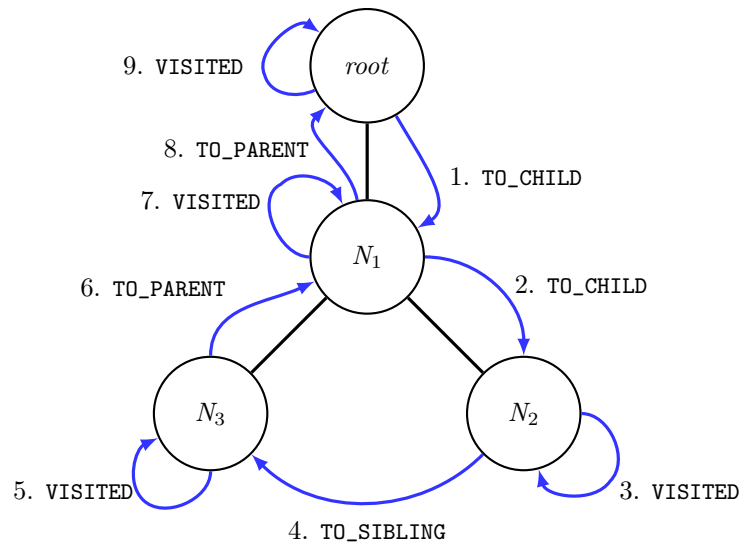


Figure 3.2: Iteration through a tree

Small-step semantics and big-step semantics

Two approaches exist in order to specify iterations (of a program). The first one is the Small-step semantics eq. (3.1) which specifies the operations of a program one step at a time [14]. Rules are defined such that from configuration c and state s , one can go to configuration c' and state s' . The idea here is to model in details the applied operations. If successive rules can be applied eq. (3.2) (hence the star above the arrow) we say that from c we can go to a *final* configuration δ with a *final* state σ .

$$\langle c, s \rangle \longrightarrow \langle c', s' \rangle \quad (3.1)$$

$$\langle c, s \rangle \longrightarrow^* \langle \delta, \sigma \rangle \quad (3.2)$$

The second approach is the Big-step semantics (eq. (3.3)) which specifies the entire transition from a configuration c and state s to a final state σ . We therefore have the equivalence (eq. (3.4)) between small steps and big steps.

$$\langle c, s \rangle \Downarrow \sigma \quad (3.3)$$

$$\langle c, s \rangle \longrightarrow^* \langle \delta, \sigma \rangle \Leftrightarrow \langle c, s \rangle \Downarrow \sigma \quad (3.4)$$

Because big steps more closely model recursive interpreters and do not have the precision of small steps, we decided to focus on the second one in order to model every move applied by the iterator. To prove properties in this semantics, we needed to define the *star* relation (\longrightarrow^*) in GALLINA² (Code 3.9).

```
Variable A: Type. (* States of the iterations *)
Variable R: A → A → Prop. (* Relation between two states. *)

(** Zero, one or several transitions: reflexive, transitive closure of [R]. *)

Inductive star: A → A → Prop :=
| star_refl: ∀ a, star a a
| star_step: ∀ a b c, R a b → star b c → star a c.

Lemma star_one: ∀ (a b: A), R a b → star a b.

Lemma star_trans: ∀ (a b: A), star a b → ∀ c, star b c → star a c.
```

Code 3.9: Definition of the transition

²Code from the course *Mechanized Semantics (MSM)* by Sandrine Blazy and David Pichardie [15]

Iterator

The traversal of the tree can be summarized by two pieces of information: the path and the last movement applied. Because we want to ensure the full coverage of the tree, we also add a third piece of information: the list of visited nodes. All these pieces of information are in the tuple: $\text{MoveSS} * \text{Path} * (\text{Visited } X)$ where X is the type of the nodes. Using Coq, we defined these rules as follows (Code 3.10).

```
(* The notation used is the following: *)
Inductive rules : A → A → Prop :=
| rule_xyz c s c' s' (* name of the rule + free variables *)
  (NAME_OF_HYPOTHESIS : first hypothesis)
  : (* ----- *)
  rules (c,s) (c',s')
.

Inductive iterator_smallstep_v X :
  Tree X → MoveSS*Path*(Visited X) → MoveSS*Path*(Visited X) → Prop :=

(* define node VISITED if we got TO_PARENT *)
| iterator_ss_v_visit_up t p visited n l
  (NODE_EXIST : getNode (0::p) t = Some (node n l))
  : (* ----- *)
  iterator_smallstep_v t (TO_PARENT,p,visited) (VISITED,p,visited)

(* define node VISITED if it has no sons *)
| iterator_ss_v_visit_no_sons t p n m visited
  (MOVE_OK : m ≠ VISITED) (* no fixpoint *)
  (MOVE_OK2 : m ≠ TO_PARENT) (* redundant but security *)
  (NODEEPPER : getNode p t = Some (node n [ ]))
  : (* ----- *)
  iterator_smallstep_v t (m,p,visited) (VISITED,p,n::visited)

(* go TO_CHILD if we did not go TO_PARENT/VISITED before *)
| iterator_ss_v_down t p n l m visited
  (DEEPER : getNode p t = Some (node n l))
  (DEEPER_NIL : l ≠ [ ])
  (MOVE_OK : m ≠ VISITED)
  (MOVE_OK : m ≠ TO_PARENT)
  : (* ----- *)
  iterator_smallstep_v t (m,p,visited) (TO_CHILD,(0::p),n::visited)

(* go TO_PARENT if we just got VISITED and there does not exist a next node *)
| iterator_ss_v_up t p n visited v l
  (NO_SIBLING : getNode ((S n)::p) t = None)
  (NODE_EXIST : getNode (n::p) t = Some (node v l))
  : (* ----- *)
  iterator_smallstep_v t (VISITED,n::p,visited) (TO_PARENT,p,visited)

(* go TO_SIBLING if we just got VISITED and there ∃ a next node *)
| iterator_ss_v_next n p t v l visited
  (TO_SIBLINGS : getNode ((S n)::p) t = Some (node v l))
  : (* ----- *)
  iterator_smallstep_v t (VISITED,n::p,visited) (TO_SIBLING, (S n)::p,visited)
.
```

Code 3.10: Rules of the Iterator using the Small-step semantics

These five rules could be summarized as:

1. If we have just gone back to the parent, the next move is VISITED.
2. If the node does not have children, the next move is VISITED.
3. If the node has a child (and the node is not VISITED), the next move is TO_CHILD.
4. If the node is VISITED and has no siblings, the next move is TO_PARENT
5. If the node is VISITED and has siblings, the next move is TO_SIBLING

Fig. 3.3 illustrates their application on the same tree as in Fig. 3.2.

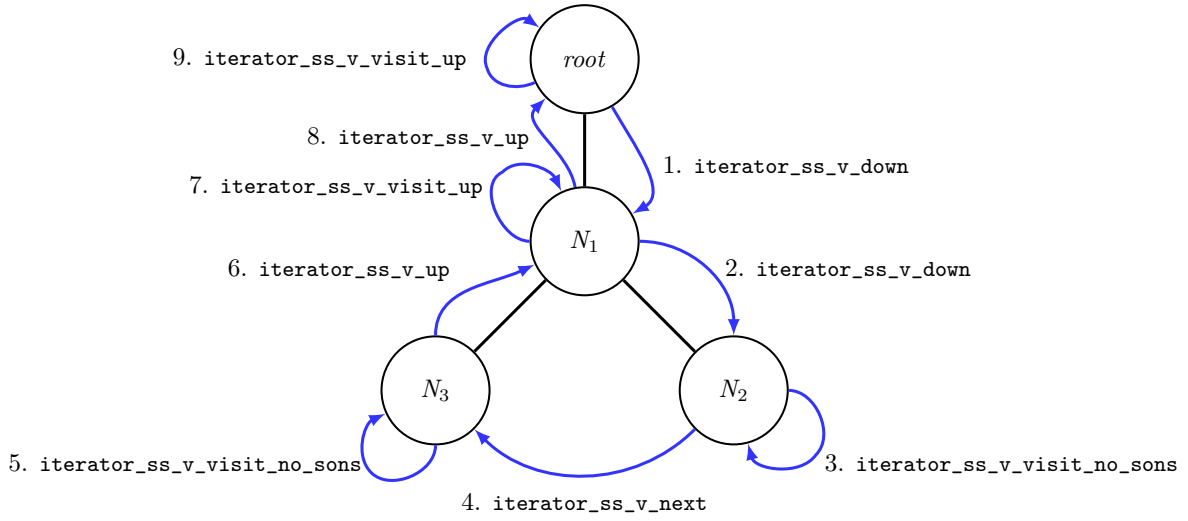


Figure 3.3: Iteration rules applied to tree traversal

3.4 Proofs of the Tree Traversal Completeness

We have specified a generic tree (Tree X), a path, the possible moves and the iterator ruling them. We are left to prove that every nodes in the tree are visited.

Iterator is Deterministic

We took the slight precaution of proving that the iterator is deterministic. In other words, in a given state, only one rule can be applied. While it is not relevant for the proof of the completeness, it helps to ensure the soundness and non-ambiguity of the iterator. This property is formalised by the lemma in Code 3.11. It generates a goal (see Code 3.12) which can be easily proven by `induction HIT1` combined with an `inversion HIT2`.

```
Lemma iterator_v_smallstep_deterministic:
  ∀ X (t:Tree X) m cs v m1 cs1 v1, iterator_smallstep_v t (m,cs,v) (m1,cs1,v1) →
  ∀ m2 cs2 v2, iterator_smallstep_v t (m,cs,v) (m2,cs2,v2) → (m1,cs1,v1) = (m2,cs2,v2).
```

Code 3.11: Iterator is deterministic

```
HIT1 : iterator_smallstep_v t (m, cs, v) (m1, cs1, v1)
HIT2 : iterator_smallstep_v t (m, cs, v) (m2, cs2, v2)
----- (1/1)
(m1, cs1, v1) = (m2, cs2, v2)
```

Code 3.12: Goal generated for the proof of determinism

Iterator is Complete

We want to prove that for any tree, the iterator goes through every node in the tree. This can be formalized as:

Iterator (TO_CHILD, empty path, empty visited list) \rightarrow^* (VISITED, empty path, set of visited nodes)

Or in Coq:

```
Corollary elemsVisited : ∀ X (tree:Tree X) m, m ≠ VISITED → m ≠ TO_PARENT →
  star (iterator_smallstep_v tree) (m, [ ], [ ]) (VISITED, [ ], (rev (vals tree))).
```

Code 3.13: The property we want to prove

where `vals tree` (Code 6.1) returns all the nodes of a tree. However this corollary is too specialized and therefore hard to prove which is why we prove a generalized version of it (Code 3.14).

```
Theorem elemsVisitedPred :  $\forall$  X (tree subtree:Tree X) m path (pred:Visited X),
  m  $\neq$  VISITED  $\rightarrow$  m  $\neq$  TO_PARENT  $\rightarrow$ 
  getNode path tree = Some subtree  $\rightarrow$ 
  star (iterator_smallstep_v tree) (m,path,pred) (VISITED,path,(rev (vals subtree)) ++ pred).
  (* pred is the list of the "already visited" nodes.*)
```

Code 3.14: Theorem: all nodes of a subtree are visited

To prove this, we have to use the induction on `getNode` defined in Code 3.7 (see p. 14) and consider two cases:

- the `subtree` has no children. Therefore we directly apply rule `iterator_ss_v_visit_no_sons` and the proof is immediate.
- the `subtree` has a list of children and we will need to prove that the resulting list of visited nodes is the concatenation of iterations on each node (which we have by induction hypothesis).

The mains steps to prove the second point are the following:

1. If we can go from state (e_1, v_1) to (e_2, v_2) and from state (e_2, v_2) to (e_3, v_3) then we can go from (e_1, v_1) to (e_3, v_3) .

$$\begin{aligned} \text{Iterator } (e_1, v_1) &\rightarrow^* (e_2, v_2) \wedge \text{Iterator } (e_2, v_2) \rightarrow^* (e_3, v_3) \\ &\Rightarrow \text{Iterator } (e_1, v_1) \rightarrow^* (e_3, v_3) \end{aligned}$$

2. If for consecutives nodes, visiting them one by one adds v_m to the list of visited nodes. Then visiting the k first nodes consecutively adds the concatenation of their respective visits to the visited list.

$$\begin{aligned} \forall k, \exists [v_0, \dots, v_n], \forall v \ m, \text{Iterator } (e_m, v) &\rightarrow^* (e_{m+1}, v_m :: v) \Rightarrow \forall k, k < n \\ &\Rightarrow \forall v, \text{Iterator } (e_0, v) \rightarrow^* (e_{k+1}, v_k :: \dots :: v_0 :: v) \end{aligned}$$

3. Specialization of 2.: if for consecutives nodes, visiting them one by one adds v_m to the list of visited nodes. Then visiting them all consecutively adds the concatenation of their respective visits to the visited list.

$$\begin{aligned} \exists [v_0, \dots, v_n], \forall v \ m, \text{Iterator } (e_m, v) &\rightarrow^* (e_{m+1}, v_m :: v) \\ &\Rightarrow \forall v, \text{Iterator } (e_0, v) \rightarrow^* (e_{n+1}, v_n :: \dots :: v_0 :: v) \end{aligned}$$

The last point (3.) provides the missing piece to prove completeness:

apply rule `iterator_ss_v_down`,
 apply 3.,
 apply rule `iterator_ss_v_up`,
 apply rule `iterator_ss_v_visit_up`

Thus I have proved the generalized version, therefore we can deduce that this iterator goes through every nodes of the tree by assuming `path = []` and `pred = []`.

3.5 Pruning and Conclusion

In the previous section, we have proven that our specification of an iterator provides a complete traversal of a tree. We now would like to tweak this iterator with an evaluation function $B : X \mapsto \text{bool}$ in order to be able to ignore some branches of the tree of type X (`Tree X`). The idea is to be able to simulate the pruning used in the scan algorithm defined in section *Differential Trails* (section 2.3). This pruning model can be applied due to the property that **adding active bits never decreases the weight of the differentials**. (cf. sub-section 2.3.3)

The iterator will be quite the same as the one defined in the previous section with the exception of some rules:

1. If we have just gone back to the parent, the next move is VISITED (*same as previous*).
2. If the node does not have children **and if $B(x)$ is true**, the next move is VISITED.
3. If the node has a child (and the node is not VISITED) **and if $B(x)$ is true**, the next move is TO_CHILD.
4. **if $B(x)$ is false**, the next move is VISITED.
5. If the node is VISITED and has no siblings, the next move is TO_PARENT (*same as previous*).
6. If the node is VISITED and has siblings, the next move is TO_SIBLING (*same as previous*).

If $B(x)$ is **true**, on the move VISITED and TO_CHILD, we add the node to the visited list. As previously to ensure some security in the specification, I proved that this second iterator is also deterministic. As the proof is very similar to the previous one, I will not present it here.

To be able to prove the completeness of the search again, we need to specify the elements we will have to visit. These are defined using this recursive function (**vals_f**, Code 6.2) similar to the one that returns the list of all nodes of a tree (**vals**, Code 6.1). The proof is the same as the one above with a small case analysis depending on the result of the function. Also by defining a function that prune the tree given a boolean function (**prune**, Code 6.3), we are able to prove an equivalence between the two results (Code 3.15). Using it we are able to prove that the iterator which traverses a pruned tree visits the same nodes as the boolean iterator traversing the full tree (Code 3.16).

```
Lemma vals_pruned : ∀ X (B: X → bool) (tree tree':Tree X),
  prune B tree = [tree'] → vals_f B tree = vals tree'.
```

Code 3.15: Equivalence of nodes

```
Theorem pruneIteration : ∀ X (B: X → bool) (tree tree':Tree X) m,
  prune B tree = [tree'] →
  m ≠ VISITED → m ≠ TO_PARENT →
  star (iterator B tree) (m, [ ], [ ]) (VISITED, [ ], rev (vals (tree')))
  ∧ star (iterator_smallstep_v tree') (m, [ ], [ ]) (VISITED, [ ], rev (vals_f B (tree))).
```

Code 3.16: Equivalence of traversal

In this chapter, I presented the formal proof assistant Coq and its language GALLINA. Using it, I have formally defined a generic tree with multiple branches and specified an iterator. Then I proved the completeness of the tree traversal. This iterator has been improved to take advantage of a function that would return whether a node (and its subtree) should be visited or not.

4. From the Code to the Proofs

One of the goals of this internship was to prove the completeness of the scan done over the differentials trails. Tools exist in order to help in such a task. The most famous ones are the *Verified Software Toolchain* (VST) [16], FRAMA-C [17] and Why3 [18]. However they are specific to the C language or variants and the program we have to verify is written in C++. Peter Schwabe¹ raised the question “*Couldn’t you just translate the program in C or did they use the dirty part of C++?*” and as a matter of fact they did as they took advantage of inheritance, classes, etc. Thus this ruled out those three candidates.

4.1 Hoare Logic

VST uses Hoare logic, a set of rules that allows to reason on programs. It is based on natural deduction and separation logic. This logic aims to describe how the execution of a piece of code changes the state of the computation. To do so, it relies on the Hoare triple notation.

$$\{P\} C \{Q\} \quad (4.1)$$

When the precondition P is met, the execution of the command C ensures the postcondition Q . In other words, if the command $\{C\}$ goes from a state s to a state s' (eq. (4.2)):

$$C : s \longrightarrow s' \quad (4.2)$$

the proof obligation formula is then eq. (4.3):

$$\forall s s', P[s] \Rightarrow C : s \longrightarrow s' \Rightarrow Q[s'] \quad (4.3)$$

However the Hoare triple notation is preferred. In Coq this could be done as in Code 4.1. The notation used is the same as the one provided by Benjamin Pierce. Most of this work on the Hoare logic has been done by a deep study of his interactive book, the *Software Foundation* [19].

```
(* defines the type assertion e.g. P[s] *)
Definition Assertion := state → Prop.

(* Use the proof obligation formula to define the Hoare tripple *)
(* " c / st ==> st' " is the property " command c goes from state st to state st' " *)
Definition hoare_triple (P:Assertion) (c:comamnd) (Q:Assertion) : Prop :=
  ∀ st st', c / st ==> st' → P st → Q st'.

(** (The traditional notation is [{P} c {Q}], but single braces
    are already used for other things in Coq.) *)
Notation "{ { P } } c { { Q } }" := (hoare_triple P c Q) (at level 90, c at next level).
```

Code 4.1: Definition of the Hoare triple in Coq

As the set of possible commands is limited: skip; sequence (commonly the “;” in C); if then else; assign (= in C); while ... , it is possible to define the behaviour of each of them with deduction rules.

¹<https://cryptojedi.org/peter/>

Sample rules are provided as follows:

- **Skip Rule** (i.e., *do nothing*)

$$\frac{}{\{P\} \text{ skip } \{P\}} \text{ (skip)}$$

- **Sequence Rule**

$$\frac{\{P\} C_1 \{Q\} \quad \{Q\} C_2 \{R\}}{\{P\} C_1 ; C_2 \{R\}} \text{ (seq)}$$

This notation is read as: "If $\{P\} C_1 \{Q\}$ (is **True**) and if $\{Q\} C_2 \{R\}$, I can infer $\{P\} C_1 ; C_2 \{R\}$." A proof is therefore built from bottom to top: if you want to prove $\{P\} C_1 ; C_2 \{R\}$, you will need to prove there exists Q such as $\{P\} C_1 \{Q\}$ and $\{Q\} C_2 \{R\}$.

- **Assign Rule**

$$\frac{}{\{Q[e/x]\} x := e \{Q\}} \text{ (assign)}$$

Where x is a variable, e is any expression. The notation $Q[e/x]$ denotes the results of substituting the term e for all occurrences of the variable x in Q .

- **Consequence Rule** (or Postcondition and Precondition weakening)

$$\frac{\{P \Rightarrow P'\} \quad \{P'\} C \{Q'\} \quad \{Q' \Rightarrow Q\}}{\{P\} C \{Q\}} \text{ (consequence)}$$

- **Conditional Rule**

$$\frac{\{B \wedge P\} C_1 \{Q\} \quad \{\neg B \wedge P\} C_2 \{Q\}}{\{P\} \text{ if } B \text{ then } C_1 \text{ else } C_2 \text{ endif } \{Q\}} \text{ (cond)}$$

- **While Rule**

$$\frac{\{B \wedge P\} C \{P\}}{\{P\} \text{ while } B \text{ do } C \text{ end } \{P \wedge \neg B\}} \text{ (cond)}$$

In *Software Foundation* [19], an *IMP* language is defined but it did not provide the `return` command and its associated semantics. As in imperative programming we use a lot of functions, I tried to define a "Return language" inspired from the work of J.-M. Madiot [20] at RU Nijmegen. After a week, we decided to drop the idea of the "Return language" for two main reasons. The first one was that the subject of the internship is not about the semantics of a language. The second reason is that we can simulate the behaviour of `return` by defining a boolean variable at the beginning of the function and using multiple `if` statements (See Code 4.2).

<pre> Definition f (x) = if (x > 1) { return; } x = x + 1; return; } </pre>	<pre> Definition f (x) = int b = 0; if (x > 1) { b = 1; } if (b == 0) { x = x + 1; b = 1; } } </pre>
---	---

Code 4.2: Example of equivalence between `return` and `if`

While it is not formally proven, a function with a `return` instruction can be expressed this way ensuring no loss of information.

4.2 Proving the Soundness of the Iterators

As the search is composed of multiple iterators, I took the simplest one as a proof of concept: the iteration on *orbitals* (defined in page 8). An orbital is composed of 4 coordinates: (x, z) the position on the column and (y_{bottom}, y_{top}) the positions of the active bits in the column. When we have 4 active bits in a single column, we consider having two orbitals. The two lower active bits are in the first one while the two upper bits are in the second one.

We can define simple properties on the coordinates:

$$0 \leq x < 5 \quad (4.4)$$

$$0 \leq z < lane_length \quad (4.5)$$

$$0 \leq y_{bottom} < y_{top} \leq 4 \quad (4.6)$$

The iterator must also iterate in the following order to cover all the orbital positions: loop on y_{top} , then on y_{bottom} , then on z and at last on x . These properties have been summarized as in Code 4.3.

```
(* specify the iterations on the y *)
Definition next_val_y (y:nat * nat) : nat * nat := match y with
| (bot,top) => if (beq_nat top 4)
then
  if (beq_nat bot 3)
  then
    (0, S 0)
  else
    (S bot, S (S bot))
  else (bot, S(top))
end.

(* specify the constraints on the y *)
Definition valid_Y (y:nat * nat) : Prop := match y with
| (bot,top) => bot < top ^ 0 ≤ bot ^ top ≤ 4
end.

(* Prove the constraints stability by the iterations *)
Theorem valid_Y_stable : ∀ b t,
  valid_Y (b,t) → valid_Y (next_val_y (b,t)).

Theorem successorOf_y_exact : ∀ bt tp,
  {{ fun st =>
    (valid_Y (st y_bottom, st y_top) ^
     bt = st y_bottom ^
     tp = st y_top) }}
  successorOf (* pseudo code equivalent of the iterator in C++ *)
  {{ fun st =>
    ( (valid_Y (st y_bottom, st y_top) ^
      (next_val_y (bt,tp)) = (st y_bottom,st y_top) ^
      st returnb = 1) ^
      st returnb = 0) }}.

Definition next_val_xz (xz:nat * nat) (y:nat * nat) (laneSize:nat): nat * nat :=
  match y, xz with
  | (bot,top),(x,z) => if andb (beq_nat top 4) (beq_nat bot 3)
  then
    if (beq_nat (S z) laneSize)
    then
      (S x, 0)
    else
      (x, S z)
    (*move to next column / row*)
  else
    (x,z)
  end.
```

```

Definition valid_Z (z l:nat) : Prop := z < l.

Theorem valid_Z_stable : ∀ x z ybt ytp l pz p,
  valid_Z z l → (next_val_xz (x,z) (ybt,ytp) l) = (p,pz) → valid_Z pz l.

Theorem successorOf_xz_exact : ∀ bt tp ix iz ls,
  {{ fun st ⇒
    (valid_Y (st y_bottom, st y_top) ∧
    ix = st x ∧ (* this is to save the initial state in the computations*)
    iz = st z ∧ (* idem *)
    bt = st y_bottom ∧ (* idem *)
    tp = st y_top ∧ (* idem *)
    st laneSize > 0 ∧
    st laneSize = ls ∧ (* idem *)
    valid_Z iz ls) }}
  successorOf
  {{ fun st ⇒ (
    (next_val_xz (ix,iz) (bt,tp) ls) = (st x,st z) ∧ (* calc and compare the values*)
    st returnb = 1 ∧
    st laneSize = ls )
    ∨ st returnb = 0) }}.

Proof.
intros.
unfold successorOf.
apply hoare_if.
- {
  eapply hoare_seq.
  eapply hoare_asgn.
  (*...*)
}
- {
  apply hoare_if.
  - eapply hoare_seq.
    eapply hoare_seq.
    eapply hoare_asgn.
    eapply hoare_asgn.
    (*...*)
}

```

Code 4.3: Stub of the proofs

The C++ code of iteration (`Orbital::successorOf()`) can be seen in Code 6.4 while its equivalent in pseudo language (`successorOf`) can be seen in Code 6.5 (both of them in Appendix).

Using the Hoare logic, We were able to formally prove the correctness of the first iterator. While it may seem easy and obvious (as in the case of the orbital it is really easy to be convinced of the code soundness), the process is really slow (about half a day is required to prove this simple function).

Conclusion

We could apply this method to the rest of the code but this would require a great amount of time. This is mainly due to the fact that I am not familiar enough with automation. The time I would have spent on more proofs would not have been compensated by the gain of the correction. With this in mind we decided to move to the second axis of the internship: "implement an abstract iterator in C++, of which the correctness is proven, provided that the instantiation respects some given properties."

5. When Proofs Provide Code

We covered the first approach (eq. (5.1)) in the previous section. Using the second one (eq. (5.2)), I will now provide an abstract iterator in which the correctness is proven, provided that the instantiation respects some given properties.

$$\textit{Specification} \wedge \textit{Software} \rightarrow \textit{Prove the correctness of the software.} \quad (5.1)$$

$$\textit{Specification} \rightarrow \textit{Software} \quad (5.2)$$

5.1 Which Language to Choose?

At the beginning of the internship, the C++ language was considered. However, looking at the difficulties to prove soundness of a given code, we decided to have an evaluation of the pros on cons of the different languages. Our requirements are the followings:

- It should be easy to learn.
- It should allow abstraction.
- If possible, it should be easy to prove.

WhyML

is a language from the *MetaLanguage* (ML) family. It provides interesting properties by adding pre- and post-conditions to functions. Moreover loops must have their invariants specified. This allows for a strong analysis. Usually using Why3 [21, 22] we can prove our properties with the assistance of other formal provers such as Alt-Ergo, Coq and many others. One of the most interesting points of Why3 is that it can produce Ocaml code and therefore provide correct-by-construction programs.

Haskell

is in the family of the functional programming language. We did not have time to research it at length: while it provides interesting capabilities, we did not have enough time to learn a new language. Therefore it was eliminated.

Ocaml

is another son of the ML family. We can extract Ocaml from Coq, it is also a possible output language of Why3. It must therefore be considered here. I have some notions of this language, moreover the syntax used by GALLINA is relatively similar. This language is used in research and in production in financial companies such as Jane Street and LexiFi. Ocaml allows Object Oriented Programming and therefore a nice layer of abstraction. I began to work on a POC in Ocaml but after some research, it seems that while it is possible to go from the proof to Ocaml, going the other way is rather difficult as it makes little sense. Using this language would also imply to later implement KECCAK- f in it.

Java

could be a nice candidate, it is Object-Oriented and may provide verification if associated with Krakatoa (Jessie and Why3)¹. While it is really close to the C++.

¹<http://krakatoa.lri.fr/>

C++

has similar advantages as Java. It is Object-Oriented and provide multi-inheritance (which Java lacks). This language is frequently used in the industry. Moreover the source code of the current scan can be directly extracted and included in the new iterator. However, should we choose that language, we would still have the initial problem of verification. Proving that some part of the code respects some specification would not be possible unless we use Hoare logic, as defined in the previous section.

C

might be another candidate. It has the same coverage as the C++ in the industry and provides similar benefits such as reusability of parts of the previous scan code. Using **VST**, we can extract lemmas and proof obligations with Hoare logic. However we aim to an polymorphic iterator. This is hard to obtain abstraction without the usage of Oriented Object. It could be done with `void*` but we do not like the idea of using *type casting*.

Gallina & Coq

are the last candidates. In the same philosophy of WhyML and Ocaml, GALLINA is a functional language, it provides abstraction and relates easily to proofs. The other pro for this language is that it follows the works of the beginning of the internship. Thus we can directly use the proofs made on the semantics trees and the iterator.

Chosen solution

With all this in mind, while WhyML would have provided a nice framework for proofs, we decided to keep up with Coq. I would write an *imperative* version of the iterator in such a way that a translation into C++ would be possible with a 1 : 1 mapping. This assumes that the algorithm is written using only simple primitives (such as `if then else`) have the same semantics in both GALLINA and C++ (see codes 5.3 and 5.5).

5.2 The Proof of Implication

We have chosen to continue with Coq and C++. As stated in the previous section, we will define the iterator in an *imperative* version. Then we will show its equivalence with the semantic iteration relations defined in Code 3.10 (page 16). In order to do so we need two functions: one to select the next move and one to apply the given move. The only piece of information we need to remember from an iteration to another, is whether or not the last move was going to the parent. Hence we will assign a boolean to `true` if the last move is `TO_PARENT`, `false` in the other cases (see Code 5.1).

```
Definition applyMove (p:Path) (m:MoveSS) : option (Path*bool) := match m,p with
| TO_CHILD, p      => Some (0::p, false)      (* go to the first child *)
| VISITED, p       => Some (p, true)          (* stay where we are *)
| TO_SIBLING, h::q => Some ((S h)::q,false)    (* go the the sibling *)
| TO_PARENT, h::q  => Some (q, true)          (* remove the head of the path. *)
(* security required by Coq *)
| TO_SIBLING, []   => None
| TO_PARENT, []    => None
end.
```

Code 5.1: Application of the selected move on a path p

We have defined how moves affect the path. Because we aim for abstraction, we define utility functions to assess properties on nodes (Code 5.2).

```
Definition NodeExists X (p:Path) (t:Tree X) : bool := ...
(* returns true if path p leads to a node *)

Definition SiblingExists X (p:Path) (t:Tree X) : bool := ...
(* returns true if path p leads to a node with a sibling *)

Definition ChildExists X (p:Path) (t:Tree X) : bool := ...
(* returns true if path p leads to a node with a child *)

Definition NodeValid X (p:Path) (t:Tree X) (f: X → bool) : bool := ...
(* if path p leads to a node n, returns f n, else returns false *)
```

Code 5.2: Utilities functions to provide pieces of informations on the tree

Using these functions, we can now define how the next move will be chosen (Code 5.3). We will not consider a recursive version where the idea would be to travel through the whole tree, but rather a single step. Hence, the implication will be easier to demonstrate. We will let the iteration (`star` relation) to the C++ part (implemented with a `while` loop).

```
(*
  what are the assumptions before going in this function? We make no such assumption.
  We only need to know one thing : was the last move TO_PARENT (last_up = true)?
*)
Definition manager X (t:Tree X) (f:X → bool) (pl:option (Path*bool)) : option (MoveSS) :=
match pl with
| None ⇒ None
| Some (p,last_up) ⇒ match getNode p t with
  | None ⇒ None
  | _ ⇒
    if andb (NodeValid p t f) (negb last_up) then (* ∧ *)
      if ChildExists p t then (* | *)
        Some TO_CHILD (* | *)
      else (* | *)
        if SiblingExists p t then (* | This part will be *)
          Some TO_SIBLING (* | directly translated *)
        else (* | into C++. *)
          Some TO_PARENT (* | *)
      else (* | *)
        if SiblingExists p t then (* | *)
          Some TO_SIBLING (* | *)
        else (* | *)
          Some TO_PARENT (* | *)
    end
end.
```

Code 5.3: Given a path we can select the next move

We have defined the `manager`, we need to prove that applying the manager on a path, implies the semantic iterator (defined in Code 3.10). This means that given a path and the possible last movement (`TO_PARENT`, `TO_CHILD`, ...), the resulting path and move verify the iterator relation (Code 5.4).

```
Theorem managerEqSemantic :
  ∀ X (B:X → bool) (tree:Tree X) (m m':MoveSS) (p p':Path) last_up last_up',
  (* we define the equivalence between the last movement and the last_up boolean value
     as hypotheses. *)
  (last_up' = true ↔ (m' = TO_PARENT)) →
  (last_up = false ↔ (m = TO_CHILD ∨ m = TO_SIBLING)) →
  (last_up = true ↔ (m = TO_PARENT) ∧ NodeExists (0::p) tree = true) →

  (* we apply the move to the path and return the boolean value for the manager *)
  applyMove p m' = Some (p',last_up') →

  (* manager hypothesis *)
  manager tree B (Some (p,last_up)) = Some m'

  →

  (* either we have an intermediate VISITED step *)
  (iterator_nv B tree (m,p) (VISITED, p) ∧ iterator_nv B tree (VISITED,p) (m', p'))
  (* Or we are right *)
  ∨ iterator_nv B tree (m,p) (m', p')).
```

Code 5.4: Theorem of the implication between the manager and the semantic iterator

Remark: because we use the `VISITED` step in the semantic definition, we have to apply the iterator twice when the first application results in a `VISITED` move. We also define the relation between the boolean and the last move as hypothesis in **Theorem** in order to specify under which conditions it holds. To prove Code 5.4, we reason by case analysis over the manager hypothesis.

5.3 The C++ Equivalence

In the previous section, we have proven an implication between a mode of iteration over a tree and a semantic definition. We now must define a tree structure that will satisfy these definitions.

From Paths and Trees to Stacks

The space of items to scan is far too vast to be built before being pruned and scanned: the number of possible differences of a $\text{KECCAK-}f[b]$ state is 2^b (with $b = 1600$ in the largest variant). While we do not need to scan all of them as we can prune with properties such as symmetry over the z -axis, the tree is still far to big to be kept in RAM.

In order to solve this problem we will consider our tree as a stack of nodes, representing at the same time the path and the current node. To do so, we assume that for any node we have the knowledge of his first child and his sibling should they exist. To retrieve these elements we define the operations `toChild()` and `toSibling()`. Our main tree is therefore recursively defined.

We defined the `Path` as a list of natural numbers `list nat` representing the index of the sibling in the list of children. When applying `TO_SIBLING` we use the successor as defined in the Peano arithmetic. This operation could be seen as applying `toSibling()` to access the sibling. And in order to get the child (when we append 0 to the path), we could consider this as applying `toChild()` to the head of the stack. Considering this model, we could therefore simplify our tree representation as a stack (Fig. 5.1).

Remarks: When we do `TO_PARENT`, the movement is equivalent to removing the head of the stack.

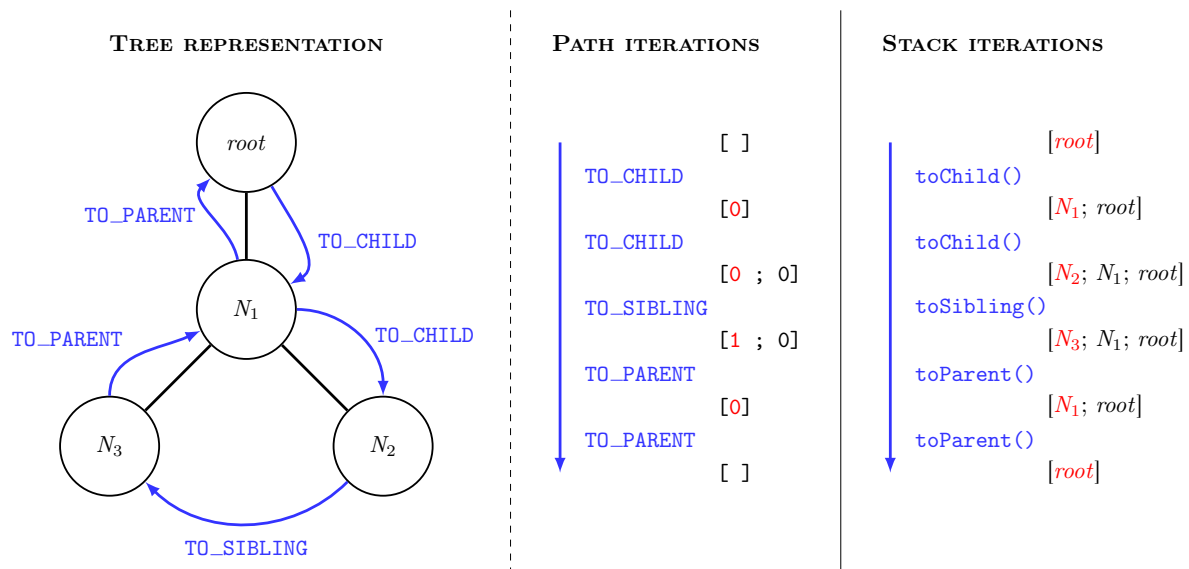


Figure 5.1: Tree/Path and Stack equivalence. The head of the list/stack is in red.

To provide a level of abstraction, we will use design patterns² such as (abstract) Factory, Decorator and Composite. We first define an `Iterable` class with a `next()` method. It aims to represent the `toSibling()`. From this `Iterable` we inherit a `MutableIterable` class, this allows us to make objects in the tree change nature (from *runs* to *orbitals*) through the iteration. Our tree (or stack) is therefore composed of `MutableIterable` objects. In order to simulate the `toChild()` movement, we use the factory, which given a context knows which kind of node to build. From this architecture we can instantiate multiple kinds of nodes whose relations are well defined. An example of instantiation of `MutableIterable` is provided in Fig. 5.2.

²<http://www.mcdonaldland.info/files/designpatterns/designpatternscard.pdf>

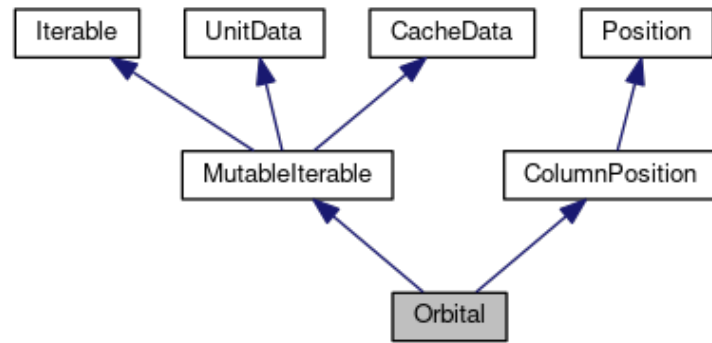


Figure 5.2: Doxygen graph for the Orbital class. Blue arrows are inheritance relations.

The Iterator: from Gallina to C++

We defined our iterator (which I named *Manager*) in a somewhat *imperative* version in GALLINA. We proved the soundness of its algorithm, we can therefore translate it into C++. We define a class *Manager* (inherited from *Iterable*). The idea here is to do `while(next()){}` where we select the move, apply it on the *Path* and return true while the traversal is not complete. The global construction could be seen in Fig. 5.3.

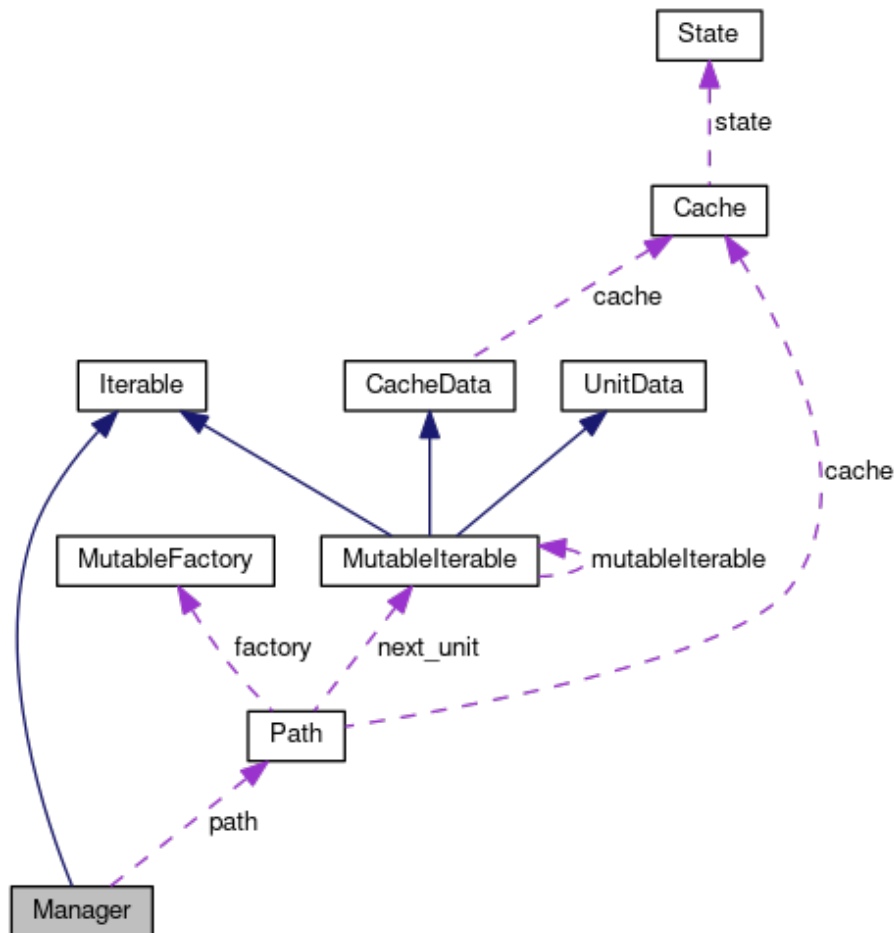


Figure 5.3: Doxygen graph for the Manager class. Blue arrows are inheritance relations, dashed purple arrows represent attributes.

The core code of the *Manager* can be found in Code 5.5 (next page).

```

/**
 * get the next_move
 * then apply it.
 */
bool Manager::next() override {
    Move m = next_move();
    return apply_move(m);
}

/**
 * The code is not optimized, it is written as defined in Coq
 */
Move Manager::next_move() {

    if (path->isNodeValid() && !is_last_move_to_parent) {
        if (path->hasChild()) {
            return T;
        }
        else {
            if (path->hasSiblings()) {
                return T;
            }
            else {
                return T;
            }
        }
    }
    else {
        if (path->hasSiblings()) {
            return T;
        }
        else {
            return T;
        }
    }
}

bool Manager::apply_move(Move move) {
    if (move == TO_CHILD) {
        is_last_move_to_parent = false;
        path->toChild();
        return true;
    }
    if (move == TO_SIBLING) {
        is_last_move_to_parent = false;
        path->toSibling();
        return true;
    }
    if (move == TO_PARENT) {
        is_last_move_to_parent = true;
        if (path->hasParent()){
            path->toParent();
            return true;
        }
        return false;
    }
    /**
     * This should never happen
     */
    printf("Manager.cpp unauthorized access ! \n");
    return false;
}

```

Code 5.5: Definition of the Manager in C++

5.4 The Chain of Trust and Conclusion

The soundness of our iterator relies on a *trusted base*, i.e. a foundation of specifications and implementations that must stay correct with respect to the specifications.

- **Calculus of Inductive Construction:** The intuitionistic logic used by Coq must be consistent in order to trust the proofs. We assumed that the functional extensionality (eq. (5.3)) was also consistent with that logic.

$$\forall A B (f g : A \rightarrow B), (\forall x : A, f x = g x) \Rightarrow f = g \quad (5.3)$$

- **Specification and Small-step semantics:** Using a semantic specification allows for a level of abstraction and provides therefore security as we focus on the meaning rather than on the precision and details of the implementation. However in order to prove the soundness of our iterator, we rely on a small-step semantics. We must be sure that we are consistent in our definitions (e.g. the deterministic proof).
- **Tree implementation and specification:** If the tree is not well defined, meaning that it represents only a subset of the trails we want to cover, then the traversal of the iterator may be complete but some trails that we wanted to cover will be missed. **This is the weakest link** of the whole proof. We move the trust from the iterator to the tree definition: the order of creations of the children and siblings must be formally defined to ensure the completeness of the search.
- **Translation from Gallina to C++:** Because we chose to work with a mapping between two languages. This assumes that the simple primitives (such as `if then else`) have the same semantics in both Gallina and C++. It also assumes the fact our translation is correct from a language to another.
- **GCC:** Because we directly translated the iterator algorithm from GALLINA to C++, we must assume that the compiled code preserves the semantics previously defined. At the time of this report (May 2016) and to my current knowledge no C++ compiler has been formally verified. It means we have no formal proof that the produced binary code satisfies our specifications.
- The last part of the trusted base is about the **Coq kernel**, the **Ocaml compiler**, the **Ocaml Runtime** and the **CPU**. These are common to all proofs done with this architecture (see section 12 in Appel et al.[2], or the fifth question in the Coq FAQ³).

In this part we have formally defined an algorithm that traverses the tree. We also proved its soundness with respect to the specifications previously defined (chapter 3). With this in mind we effectively decreased the size of the untrusted part and moved it to the tree definition.

³<https://coq.inria.fr/faq>

6. Conclusion

In this report I provided a brief introduction to differential cryptanalysis. I also explained the concept of trails and weight. I tried to provide explanations of how the search is conducted which justifies the semantic definition of the iterator and its soundness. Then I was able to explore the two axes proposed. I showed that proving the current code is possible using Hoare logic, however it would require a considerable amount of time. Extracting a working code from specification is faster by a large margin. With this second method I was able to produce an abstract iterator whose tree traversal is proven complete.

Regarding our initial objective, we were not able to provide a stronger confidence on the previously calculated bounds. However, the proven iterator contributes to the trust in later implemented searches.

During this internship I learned a lot about Hoare logic, being fairly new in the domain and I still think I have a lot to learn about it. Also in our Mechanized Semantics course, we were able to familiarize ourselves with Coq. But in most of our proofs we only had to find the right order of instructions (*induction, inversion...*) and thus we were restricted in a somewhat predefined path. To prove the complete tree traversal, I needed to think out of the box and lead my own proof rather than just being guided.

As a final words, I would say that over the many things I learned during these last two years, the many courses taken at the *Institut National de Sciences Appliquées (INSA)* and during my Master degree, I do not see any knowledge that I did not use here.

Bibliography

- [1] A. J. Menezes, P. C. Van Oorschot, and S. A. Vanstone, *Handbook of applied cryptography*. The CRC Press series on discrete mathematics and its applications, 2000 N.W. Corporate Blvd., Boca Raton, FL 33431-9868, USA: CRC Press, 1997.
- [2] A. W. Appel, "Verification of a cryptographic primitive: SHA-256," *ACM Transactions on Programming Languages and Systems*, vol. 37, pp. 7:1–7:??, Apr. 2015.
- [3] J. Daemen and G. Van Assche, "Differential propagation analysis of keccak," in *Proceedings of the 19th International Conference on Fast Software Encryption, FSE'12*, (Berlin, Heidelberg), pp. 422–441, Springer-Verlag, 2012.
- [4] E. Biham and A. Shamir, "Differential cryptanalysis of DES-like cryptosystems," Technical report CS90-16, Department of Computer Science, Weizmann Institute of Science, Rehovot, Israel, July 1990.
- [5] G. Bertoni, J. Daemen, and M. Peeters, "Cryptographic sponge functions," report, STMicroelectronics, Antwerp, Belgium (??), Jan. 2011.
- [6] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche, "The KECCAK reference," January 2011. <http://keccak.noekeon.org/>.
- [7] D. Cox, J. Little, and D. O'Shea, *Ideals, Varieties, and Algorithms: An Introduction to Computational Algebraic Geometry and Commutative Algebra*. Undergraduate Texts in Mathematics, Springer New York, 2010.
- [8] J. Daemen and V. Rijmen, *The design of Rijndael: AES — the Advanced Encryption Standard*. Berlin, Germany / Heidelberg, Germany / London, UK / etc.: Springer-Verlag, 2002.
- [9] FIPS, *Advanced Encryption Standard (AES)*. National Institute for Standards and Technology, Gaithersburg, MD 20899-8900, USA, Nov. 2001.
- [10] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche, "On alignment in KECCAK," in *ECRYPT II Hash Workshop 2011*, 2011.
- [11] "The coq proof assistant - Reference Manual." <https://coq.inria.fr/refman/>.
- [12] A. Chlipala, *Certified Programming with Dependent Types - A Pragmatic Introduction to the Coq Proof Assistant*. MIT Press, 2013. <http://mitpress.mit.edu/books/certified-programming-dependent-types>.
- [13] D. Larchey-Wendling, "Re: [coq-club] how to prove an inductive property on trees.." <https://sympa.inria.fr/sympa/arc/coq-club/2016-02/msg00042.html>.
- [14] A. Myers, "IMP: Big-step and small-step semantic." <http://www.cs.cornell.edu/courses/cs6110/2009sp/lectures/lec05-fa07.pdf>.
- [15] S. Blazy and D. Pichardie, "MSM: Mechanized semantics." <http://master.irisa.fr/index.php/en/prog-ue-en/108-programme-en/modules-en/247>.
- [16] "Verified Software Toolchain." <http://vst.cs.princeton.edu>.
- [17] "Frama-c." <http://frama-c.com>.
- [18] "Why3." <http://why3.lri.fr>.

- [19] B. C. Pierce, C. Casinghino, M. Greenberg, V. Sjöberg, and B. Yorgey, *Software Foundations*. University of Pennsylvania, 2011. <http://www.cis.upenn.edu/~bcpierce/sf/>.
- [20] J. M. Madiot, “Specification of imperative languages using operational semantics in coq.” [Report](#), [slides](#).
- [21] F. Bobot, J.-C. Filliâtre, C. Marché, and A. Paskevich, “Why3: Shepherd your herd of provers,” in *Boogie 2011: First International Workshop on Intermediate Verification Languages*, (Wrocław, Poland), pp. 53–64, August 2011.
- [22] J.-C. Filliâtre and A. Paskevich, “Why3 — where programs meet provers,” in *Proceedings of the 22nd European Symposium on Programming* (M. Felleisen and P. Gardner, eds.), vol. 7792 of *Lecture Notes in Computer Science*, pp. 125–128, Springer, Mar. 2013.

Global Internship Development

One could say this internship began in **October 2015** when we were tasked to do a bibliographic report on the subject. This report made us investigate on the subject to get the required knowledge on the domain. I learned Differential Cryptanalysis and Linear Cryptanalysis during this time.

February

The first month was dedicated to a formal approach: getting in the subject, understanding what are the expectations, presenting to my supervisors CoqIDE and how a proof works. I also wrote the specification of the semantic tree and iterator semantic. I then proved its soundness. At the end of February, I was able to make a quick presentation of my work: "Semantic Iterator over a Semantic Tree, Proof of complete path".

March

While the proof was being completed in Intuitionistic Logic using Natural Deduction rules, they asked me to provide them with a presentation about the logic used in formal methods. This second assignment allowed me to broaden my knowledge in the domain and get a better understanding of the philosophy behind it. I therefore did a second presentation: "A brief introduction to Logic and its applications. Classical, Intuitionistic and Hoare" mainly focused on the history (Hilbert, Gödels, Church, Gentzen and Hoare).

Then I was tasked to prove the current implementation of the iterators. I did research about **VST** [16] and other tools. In the paper I read, I was advised to improve my knowledge about Hoare Logic and how it is applied in Coq. Therefore I spent time on Software Foundation by Benjamin C. Pierces [19]. Seeing that we did not have the possibility to use verification tools such as **VST** or Frama-C, I tried to define a semantic over a toy language with the **return** instruction (namely the **RETURN** Language, and extension of the **IMP** language.) That was a failed attempt. Maybe if I was using monads (which I discovered later) things might have been slightly different.

At the same time, I worked on trying to provide a formal proof in the PhD of Silvia Mella over the *Z-canonicity*. But as it was not the subject of this internship, I preferred to switch back to my original tasks.

April

At the beginning of April, I was able to prove that the Orbital iterator had a complete traversal. Given the time spent on proving a single and simple version of the iterators, we decided to switch to the second axis of this internship: Provide a generic iterator whose coverage is proven while its instantiation respects some constraints. I was asked to provide a choice of language and motivate which one we should use. As explained in section 5.1, we chose to go with Coq and a 1 : 1 translation into C++.

May

From my previous experiences, time to write a report can be often underestimated. For this reason, I decided to dedicate May to the writing of this report and to prepare the slides for the defense.

June

The deadline to submit this report is on the third. From the fifth to the eleventh, I will be attending a Summer school on “real-world crypto and privacy” in Šibenik¹ (Croatia). It is organized by Radboud University² (Netherland), KU Leuven³ (Belgium), ETH⁴ & ZISC Zurich⁵ (Switzerland) and FER⁶ (Croatia). One of my co-supervisors, Joan DAEMEN will be teaching there.

On the twentieth, I will be defending this report at the INSA for an engineering diploma in Computer Science. And on the twenty-third, I will be defending this report for my *Research Master’s degree in Computer Science (MRI)*. And finally from the twenty-seventh to the first of July, I will be attending a second summer school (EJCP⁷) dedicated to formal methods in Lille⁸. It is organized by the CRISTAL⁹ Laboratory.

¹<http://summerschool-croatia.cs.ru.nl/2016/>

²<http://www.ru.nl>

³<http://www.kuleuven.be>

⁴<https://www.ethz.ch>

⁵<https://zisc.ethz.ch>

⁶<http://www.fer.unizg.hr>

⁷École jeunes chercheurs en programmation

⁸<http://ejcp2016.univ-lille1.fr>

⁹<http://cristal.univ-lille.fr>

Pieces of code

```
Fixpoint vals X (t:Tree X) : list X := match t with
| node n l => n :: (flat_map (vals X) l)
end.
```

```
Arguments vals [X] _ _.
```

Code 6.1: recursive function returning all nodes of a tree as a list

```
Fixpoint vals_f X (b:X → bool) (t:Tree X) : list X := match t with
| node n l => if (b n) then n :: (flat_map (vals_f X b) l) else []
end.
```

```
Arguments vals_f [X] _ _.
```

Code 6.2: recursive function returning all nodes of a tree as a list satisfying a boolean function

```
Fixpoint prune X (B: X → bool) (tree:Tree X) : list (Tree X) := match tree with
| node n l => if (B n) then [node n (flat_map (prune X B) l)] else []
end.
```

```
Arguments prune [X] _ _.
```

Code 6.3: recursive function pruning a tree keeping the nodes satisfying a boolean function

```
//This method adapts the orbital with respect to f_pos=y+5(z+wx)
// and f_order=2^25w\sum{2^{-f_pos}}
bool Orbital::successorOf(unsigned int laneSize)
{
    // if possible, newOrbital has the same y_bottom and higher y_top
    if (y_top < 4) {
        y_top++;
        return true;
    }
    else
    {
        // else, if possible, newOrbital has higher y_bottom
        if (y_bottom < 3) {
            y_bottom++;
            y_top = y_bottom+1;
            return true;
        }
        else
        {
            // else, if possible, newOrbital is in the next column in the same sheet.
            if (z < laneSize-1) {
                z++;
                y_bottom = 0;
                y_top = 1;
                return true;
            }
            else

```

```

    {
        // else, if possible, newOrbital is in the next column in the first slice
        if (x < 4) {
            x++;
            z=0;
            y_bottom = 0;
            y_top = 1;
            return true;
        }
        else
            return false;
    }
}
}
}
}

```

Code 6.4: Definition of the next position for an Orbital

```

Definition successorOf :=
If BLt (AId y_top) (ANum 4)
then
    y_top ::= APlus (AId y_top) (ANum 1) ;;
    returnb ::= ANum 1
else
    If BLt (AId y_bottom) (ANum 3)
    then
        y_bottom ::= APlus (AId y_bottom) (ANum 1);;
        y_top ::= APlus (AId y_bottom) (ANum 1) ;;
        returnb ::= ANum 1
    else
        If BLt (AId z) (AMinus (AId laneSize) (ANum 1))
        then
            z ::= APlus (AId z) (ANum 1);;
            y_bottom ::= ANum 0;;
            y_top ::= (ANum 1) ;;
            returnb ::= ANum 1
        else
            If BLt (AId x) (ANum 4)
            then
                x ::= APlus (AId x) (ANum 1);;
                z ::= ANum 0;;
                y_bottom ::= ANum 0;;
                y_top ::= (ANum 1) ;;
                returnb ::= ANum 1
            else
                returnb ::= ANum 0
            fi
        fi
    fi
fi.

```

Code 6.5: Equivalence of the successorOf in IMP language

Résumé

La cryptanalyse différentielle (DC) est une discipline qui cherche à exploiter les probabilités de propagations de différences dans le but de casser des itérations de primitives cryptographiques. La version simple utilise des chemins différentiels qui consistent en une séquence de différences au travers des rondes de la primitive. Étant donné un tel chemin, il est possible d'estimer la probabilité différentielle (DP), autrement dit la proportion de toutes les paires possibles ayant pour différence initiale la première différence du chemin et qui montrent les différences intermédiaires et finale lors des différentes rondes. Cette probabilité peut être mise en relation avec un poids. Afin d'établir la sécurité d'un algorithme, on peut chercher une borne inférieure à ce poids. Plus cette dernière sera haute, plus la primitive sera sûre.

Les bornes inférieures sur la cryptanalyse linéaire et différentielle ont été prouvées pour KECCAK- f [25] à [200] et pour KECCAK- f [1600] (DC seulement) [Daemen and Van Assche, FSE 2012].

Ces bornes ont été obtenues à l'aide d'un ordinateur. L'idée étant de parcourir l'intégralité de l'espace des chemins jusqu'à un poids T . Si un chemin avec un poids minimum est trouvé, on a une borne exacte. Dans le cas contraire, la borne inférieure est au moins $T+1$.

La correction de la preuve dépend de la correction du parcours. En particulier, on doit s'assurer de la correction de l'élagage des branches et de l'implémentation. En d'autres termes, l'algorithme de recherche fait partie de la "preuve" qu'il n'existe pas de chemin différentiel en dessous d'un certain poids. De fait, faire confiance à la preuve implique de vérifier le programme.

Ce rapport présente comment les méthodes formelles peuvent aider à s'assurer de la complétude de la recherche. Il propose un début de la preuve de la correction du code. Il fournit également une spécification formelle d'un itérateur et d'une implémentation générique qui pourra être utilisée pour visiter intégralement des arbres.

Abstract

Differential cryptanalysis (DC) is a discipline that attempts to find and exploit predictable difference propagation patterns to break iterative cryptographic primitives. The basic version makes use of differential trails (also called characteristics or differential paths) which consist of a sequence of differences through the rounds of the primitive. Given such a trail, one can estimate its differential probability (DP), namely, the fraction of all possible input pairs with the initial trail difference that also exhibits all intermediate and final differences when going through the rounds. This probability is related to a weight. The idea is to provide a bound to this weight: the higher the more secure the algorithm will be.

Lower bounds on differential and linear trails have been proven for KECCAK- f [25] to [200] and for KECCAK- f [1600] (DC only) [Daemen and Van Assche, FSE 2012].

These bounds are obtained by computer-assisted search. The idea is to scan the entire space of all trails up to a given weight T . If a trail with minimum weight is found, we get a tight lower bound. If no trail is found, then $T+1$ is a lower bound, though it is untight.

The correctness of the proof depends on the correctness of the search. In particular, we must ensure the correctness of the lower bound of the branches we cut, as well as the correctness of the implementation. In other words, the search program is part of the "proof" that there are no trails below some given weight, so believing the proof implies verifying the program.

This report presents how formal methods can help to ensure that the search is complete. It provides a start for a formal proof of the soundness of the code. It also provides formal specification of an iterator and a generic implementation which could then be used to completely traverse a tree.