UMinho | 2012

Luís Paulo Ferreira Miranda **Domain-specific Languages for Cryptographic Software**

Universidade do Minho
Escola de Engenharia

Luís Paulo Ferreira Miranda

Domain-specific Languages for Cryptographic Software

Janeiro de 2012

Universidade do Minho
Escola de Engenharia

Luís Paulo Ferreira Miranda

Domain-specific Languages for Cryptographic Software

Tese de Mestrado
Mestrado em Engenharia Informática

Trabalho efectuado sob a orientação do
Professor Doutor José Carlos Bacelar Almeida

Janeiro de 2012

Abstract

Cryptography plays an important role in our society, essentially because it is used in critical computer systems that must work properly, even in the face of errors or human mistakes. Banking or health care are examples of areas which use hardware and software that must work in every situation. The main goal of the use of cryptography in those systems is to achieve information security, which in most cases is sensitive.

In the last few years, programming languages that focus on a particular domain emerged, denominated as Domain-specific Languages (DSLs). Two DSLs for cryptography appeared, Cryptol and CAO, both aiming to increase the productivity of developers, but also to improve the communication between them and domain experts.

Cryptol is a functional language and has an associated toolkit composed by a verification suite and compilation back ends to languages such as C or VHDL (a hardware description language). CAO is an imperative language, with a syntax similar to C's and also has an associated toolkit with tools that allows, for instance, the introduction of higher-order operations into the language.

In this work, these two DSLs for cryptography will be analysed, focusing on its features and how they can re-target a published algorithm into a specific implementation. Furthermore, it was developed a compiler tool that aims to translate CAO source code into Cryptol, in order to compile it afterwards to VHDL.

Finally, a case study focusing on elliptic curve cryptography, was used to compare the two DSLs and to test the developed tool.

Resumo

A criptografia desempenha um papel importante na nossa sociedade, visto que é utilizada em sistemas de computação designados como críticos, que têm que funcionar mesmo na presença de erros. Áreas como os sistemas bancários ou de saúde usam software e hardware, que têm que funcionar em todas as circunstâncias. O principal objectivo para o uso de criptografia nesses sistemas, é o de garantir a segurança da informação, que em muitos casos é sensível.

Nos últimos anos, foram surgindo linguagens de programação que se focam num domínio específico, chamadas de linguagens de domínio específico (DSLs). No domínio da criptografia, apareceram as linguagens Cryptol e CAO, ambas ambicionando aumentar a produtividade dos programadores, mas também aumentar a comunicação entre estes e os especialistas do domínio.

O Cryptol é uma linguagem funcional e tem um conjunto de ferramentas associadas, compostas por um conjunto de ferramentas de verificação e de compilação para linguagens como C ou VHDL, que é uma linguagem descritiva de hardware. O CAO é uma linguagem imperativa, com uma sintaxe idêntica à do C, e tem também um conjunto de ferramentas associado, que permite a introdução de operações de alto nível na linguagem, por exemplo.

Neste trabalho, essas duas linguagens foram abordadas, em particular as suas funcionalidades, e como podem ser usadas para implementar um algoritmo através da sua especificação. Além disso, foi desenvolvida uma ferramenta de compilação que pretende transformar código fonte CAO em código Cryptol, de forma a compilá-lo para VHDL posteriormente.

Por fim, um caso de estudo que foca curvas elípticas para criptografia, foi utilizado para comparar as duas DSLs e também para testar a ferramenta desenvolvida.

Acknowledgements

First of all I would like to thank my supervisor Professor José Carlos Bacelar for all the support, research guidance and all the suggestions through this work.

Many thanks also to all the people involved into the SMART project at the Minho's University, in particular Manuel Barbosa, José Barros, Manuel Alcino, Paulo Silva, Diogo Moreira and Tiago Oliveira for all the interesting discussions and for the good working environment provided. During the execution of this work, I was given a research grant - Bolsa de Investigação Científica (BIC), funded by the ENIAC JU program (GA 120224).

I want to give a special thank you to all my friends for all the encouragement and support, in particular the ones from my hometown and Confraria do Matador. At the same time I want to apologize for not being so present in the past months. A special thank you for Rita, for all the kind of support and patience throughout this work.

Last but not least, thanks to my parents and my brother, for the constant encouragement and for always having supported me throughout my life.

“Luck is where preparation meets opportunity.”
- Randy Pausch, The Last Lecture

Contents

List of Figures

List of Tables

Listings

Acronyms

AES Advanced Encryption Standard

AST Abstract syntax tree

CFG Control-flow Graph

CACE Computer Aided Computer Engineering

DSL Domain-specific Language

ECDSA Elliptic Curve Digital Signature Standard

FPGA Field-programmable Gate Array

IT Information Technologies

MACs Message Authentication Codes

VHDL VHSIC Hardware Description Language

SSA Single-static Assignment

SMART Secure Memories and Applications-related Technologies

VCGen Verification Condition Generator

Chapter 1

Introduction

Nowadays, information systems are important in our society. These systems use a wide range of electronic devices: personal computers, mobile phones or other embedded systems. Everyday, these systems generate and store big amounts of data and, as most of them are network-based, a great amount of traffic is transmitted through the network.

Complex computer systems in specific areas like banking, health care or aeronautics usually are considered to be critical systems. The information handled by these systems is sensitive. For example: in the banking systems there are managed bank accounts, client data and money transactions; in the health care systems there are managed patients' medical records, diseases and other private information to doctors and the patient.

The design of these systems must have a different approach from typical software engineering, approach in which developers keep programming and tuning the system until it does what it is supposed to do. In a critical system, security requirements are of major priority: these systems have to work in every single situation, even when concerning human mistakes, arbitrary errors or in the presence of a malicious adversary.

When properly used, cryptography may assure information security. The formal mathematical nature of it applied to computer science and electronic engineering can be used to achieve security properties such as confidentiality, data integrity or user authentication. In fact, cryptographic components are increasingly being incorporated into electronic devices and play a big role in most part of Information

Technologies (IT) software. Therefore, since these components can be used to assure security, they assume an important part in their systems.

At a software level, cryptography may be implemented using any general purpose language such as JAVA or C/C++. However, in the past few years, two domain-specific languages (DSLs) for cryptography appeared, capturing the semantics of the cryptographic domain: Cryptol and CAO. Both languages incorporate features that allow one to represent cryptographic primitives, such as big numbers, number theory libraries or bit-wise operations. The DSLs are smaller when compared to general purpose languages, they make their programs easier to write, to reason about and to maintain.

The Cryptol language was designed by Galois and it is based on the functional programming paradigm. It allows developers to design high-level specifications of cryptographic algorithms and takes advantage of polymorphic functions, lambda abstractions, data records, and sequence comprehension. The Cryptol language is part of a toolkit which is composed by the language itself and an interpreter to evaluate produced code. Furthermore, it includes back end compilers to other languages like C or VHSIC Hardware Description Language (VHDL), which can be used in electronic design automation to describe digital and mixed-signal systems. Cryptol also includes formal methods-based tools to generate formal proofs, to be tested and verified in external theorem provers such as static code analysers. In particular, the Cryptol-VHDL compiler is an interesting tool, as it allows developers to design high-level specifications of cryptographic algorithms based on published standards and then compile them to VHDL, which is an industry-standard chip layout language or a Field-programmable Gate Array (FPGA) circuit.

The CAO language was designed by the Cryptography and Information Security Group of the University of Bristol and it was designed to allow the implementation of cryptographic primitives such as block ciphers, hash functions or sequences of finite field arithmetics. The language is based on the imperative programming paradigm and has a syntax similar to C's. In the context of the Computer Aided Computer Engineering (CACE) project, a set of tools were developed in order to improve CAO's functionalities, such as the CALF compiler or the CAO-SL: the former introduces higher-level features into the language such as polymorphism and dependent types; the latter allows one to define contracts such as pre-conditions, post-conditions and loop invariants.

1.1 Objectives

The main purpose of this work, is to study, analyse and compare the two DSLs. The inherent differences of both languages start on their paradigms. The constructors and features that can be found in an imperative general purpose language such as JAVA or C++, when compared to a functional one like Haskell, are clearly distinct, requiring a different programming style. As cryptographic algorithms are published mostly as pseudo-code, the first part of this dissertation will focus on the comparison between two implementations of an widely used algorithm: the AES. Some key aspects like the type systems, iteration building or the manipulation of containers and structures of data have different approaches both in functional and imperative paradigms, and will be addressed further in this document.

One of the most interesting features of the Cryptol toolkit is the VHDL back end. However, this particular Cryptol back end has some limitations, and some of them are hardware related. Taking this into account, this dissertation proposes a tool, CAO2Cryptol, that aims to take CAO source code as input, and as result generate the equivalent Cryptol source code. Particularly, the output of the compiler must be translatable to a hardware description language, such as VHDL.

Due to the differences of the paradigms of Cryptol and CAO, in the construction of the CAO2Cryptol compiler, some components of the CAO language such as the Abstract syntax tree (AST) were used to generate a Control-flow Graph (CFG), a graph structures that indicates the program flow. Consequently, compiler related algorithms such as the Dominance Frontier algorithm were applied to the CFG, in order to obtain the SSA form of a given program. After a set of transformations over the input program, a source code program written in CAO, can be translatable to Cryptol, and it is expected that the output to be compilable to VHDL.

1.2 Dissertation Outline

This work is divided into six chapters. In the current chapter, we present an introduction to all the work related to this dissertation.

Chapter 2 describes the domain-specific languages focused on the cryptographic domain: Cryptol and CAO, presenting the features of both languages and their toolkits.

Chapter 3 introduces the AES algorithm and discusses its implementation both in Cryptol and CAO. The implementations of the algorithm will be compared, focusing on particular key aspects of the languages.

Chapter 4 analyses the compilation of the AES algorithm to VHDL, through the Cryptol toolkit. It is used the AES-128 version of the algorithm, which uses a 128-bit key to encrypt data. The 32-bit implementation of AES algorithm will be also addressed in this chapter, as well as the algorithm approach that uses auxiliary tables (T-boxes) to optimize the algorithm.

Chapter 5 presents and describes the building process of the CAO2Cryptol compiler, oriented on hardware compilation. The limitations of the Cryptol toolkit and the requirements for the compiler will be presented, followed by the steps required to transform CAO source code into Cryptol source code.

Chapter 6 evaluates the implemented compiler with a case study. Elliptic curve cryptography primitives are used to test the CAO2Cryptol compiler, focusing on the operations which manipulate elliptic curve points.

In the last chapter we present the conclusion of this dissertation, summarizing the developed work and describing what can be done in the future based on it.

Chapter 2

Domain-specific languages for cryptography

The use of representation models, specification techniques or programming languages to solve problems of a particular domain has risen in the past years. According to [Fow10], a Domain-specific Language (DSL) may be defined as a *computer programming language of limited expressiveness focused on a particular domain*, and can be defined by the following key elements:

- **Computer programming language**
- **Language nature**
- **Limited expressiveness**
- **Domain focus**

Like any general purpose programming language, a DSL must be human readable and machine processable, allowing humans to interact with computers in a fluent way. Since they are domain-specific, these languages must focus in the core elements of the target domain, supporting the required language features to solve domain's problems [Fow10].

DSLs are popular mainly because they increase the productivity of developers and improve the communication between them and domain expertises [Fow10]. They are used in a large set of contexts. For instance, SQL is an example of a DSL, which focuses on relational databases queries: it captures the semantics of

the problems of that domain and provides a structured language, so that developers may deal with and manipulate relational databases. Regular expressions, CSS or XML configuration files such as Struts and Hibernate are other examples of DSLs, all of them human readable and focused on their domains [Fow10].

In this chapter, we will focus on the cryptographic domain. The main components of cryptographic protocols will be exposed and analysed in order to identify which elements must be present in a domain-specific language designed to capture the semantics of the cryptographic domain. Later in this chapter we will present Cryptol and CAO, two DSLs for cryptography. Each DSL is also complemented with a toolkit, providing auxiliary tools like verification and re-targeting compilers, useful for program validation and to the compilation of cryptographic code to other languages, respectively.

2.1 Cryptography Domain

Cryptography may be seen as a mixture of concepts from different backgrounds. It takes advantage of knowledge from areas such as mathematics, applying them to software and hardware, which requires an additional knowledge of computer science and electronic engineering [BMP+11]. As described in [VOMV96], *cryptography is the study of mathematical techniques related to aspects of information security such as confidentiality, data integrity, entity authentication and data origin authentication*. In fact, cryptography aims to accomplish other security goals such as authentication, identification, authorization, validation, access control, certification or time stamping [VOMV96]. More specifically, cryptography boils down to information security and secure communication between agents, which can be people, computers or other embedded systems.

Confidentiality aims to guarantee that only authorized agents may access secret information [VOMV96]. Data integrity assures that data is not manipulated by unauthorized parties, ensuring that no malicious adversaries insert, delete or substitute existing data by incorrect or corrupted values [VOMV96]. Authentication is directly related to identification: it is the process in which an agent proves to another that it really is who it claims to be [VOMV96][Sch93]. Finally, non-repudiation aims to prevent that an agent can not deny previous actions or commitments [Sch93]. There are a lot of scenarios in which cryptography is used to

achieve information security or access control of any type of resource, requiring the use of the appropriate techniques to assure that people's privacy, for instance, is not violated.

Confidentiality The main building block to achieve confidentiality are ciphers, which use a secret usually referred as the key, to assure that only authorized agents have access to private information [Sch93]. Ciphers are divided into two major groups: symmetric and asymmetric, both aiming to create unintelligible messages to whoever does not know the secret/key. Symmetric ciphers use the same key for encryption and decryption [Sch93]. Furthermore, symmetric ciphers may be either stream ciphers or block ciphers: the formers create an extended version of the key based on the original one, combining with the plain text, on bit at a time; the latter split the original plain text into smaller blocks, performing the encryption or decryption operation on these blocks.

Data Integrity To assure data integrity, one must guarantee that a received message was not modified since it has been sent. The basic build block to achieve integrity are hash functions, which produce a message digest, a smaller bit string known as the hash value [Sch93]. Most hash functions are one-way, i.e. it is hard to retrieve the original message based on the hash value [Sch93].

Authentication and Identification In some scenarios, it is also important to authenticate a message or the entity, i.e. to verify that the sender is who it claims to be [Sch93]. Based on asymmetric cryptography, digital signatures may be used to authenticate an agent: the sender signs the message with its private key and the receiver(s) verify the authenticity of the signature by using the sender public key [Sch93]. In some cases, it important to guarantee the authentication of a message and its integrity, which can be achieved by the use of Message Authentication Codes (MACs). These MACs take advantage of symmetric cryptography by combining a symmetric cipher and a hash function.

Non-repudiation Non-repudiation is used to prevent that an agent later denies some action or commitment, such as sending a message [VOMV96][Sch93].

Non-repudiation is also achieved by the use of digital signatures or even time stamping techniques [VOMV96].

At the core of these cryptographic primitives (symmetric or asymmetric ciphers, hash functions, MACss, digital signatures, etc.), there are protocols which use algorithms from fields such as information theory, complexity theory, number theory or abstract algebra, most of them implemented through low-level mathematical primitives [VOMV96]. However, the natural representation of data in a computer system is through bits, making bit-wise operations, such as rotates, shifts and permutations, widely used to represent and manipulate cryptographic algorithms. Therefore, a DSL for the cryptographic domain must assure that these primitives are included in the language's features somehow, allowing a re-targeting from the specification of a cryptographic algorithm to a concrete implementation in a programming language.

2.2 Cryptol

Cryptol was designed by Galois[1] in parcery with cryptography experts from the National Security Agency (NSA) and is a pure, declarative and functional language [cry08]. It brings a formal methods-based approach to the cryptographic domain and allows cryptography developers to produce code in a clear and direct way [cry08]. Besides the programming language, Cryptol is a toolkit composed by a set of compilation back ends to languages such as C or VHDL. It also may be seen as a framework for verification [cry08].

With this tools, Cryptol may be seen as a language for cryptography, allowing developers to implement cryptographic algorithms, and re-target them to software and hardware platforms [cry08]. It allows one to implement cryptographic algorithms based on the published standard, verify it through the verification suite (e.g. by checking the equivalence of a high-level specification against test vectors) and then compile it to a target platform such as C (for software) or VHDL (for hardware). Particularly, the Cryptol language and the generated VHDL code maybe useful to design a FPGA board circuit or to use it on embedded systems or smart cards. [cry08].

[1] www.galois.com

2.2.1 Type system

Cryptol is a strongly typed programming language and has a sophisticated type system [cry08]. Similar to what happens in functional languages such as Haskell, it allows developers to implement abstract functions by using lambda expressions or by declaring polymorphic functions. It supports higher-order functions, nested functions and has the Bit type as its main building block, assuming the values `True` (1) or `False` (0).

The natural way of structuring bits is via sequences. Sequences are ordered and indexed structures of data, where all elements have the same type. This allow one to build complex structures such as numeric literals, vectors, matrices, etc. For example, an 8-bit word (Byte) may be represented by a sequence with eight elements in which each element is a Bit and is represented by its word size: [8].

When declaring a function, the function signature may be specified, i.e. the type of each argument and the output may be specified. Although the function signature may be omitted, it is considered a good practice to write it. For example, the identity function may be defined in Cryptol as follows:

Listing 2.1: Identity function in Cryptol

```
1 f : {a} [a] -> [a];
2 f x = x;
```

In this example, it is explicitly defined that the input and the output have the same type `[a]`. Additionally, this function `f` is polymorphic: for any word size {*a*}, it takes an *a*-sized word and returns an *a*-sized word [cry08]. However, some restrictions may be applied in the function signature. For instance, if the function signature is substituted by `f : {a} [a > 6] => [a];`, `f` is still polymorphic, however it will only accept as input an *a*-sized word with more than 6 bits.

Another way to structure data is by using tuples, that unlike sequences, do not need to have the same type in all of their elements [cry08]. Consider the following example, that aims to store a bit string with a byte and one with 1 bit:

Listing 2.2: Tuple representation in Cryptol

```
type NewType = ([8],[1]);

t : NewType;
t = (15,1);
```

The two elements have different types, which makes this not possible to be represented using sequences.

Additionally, the Cryptol language has a record system which extends tuples with two additional features: the fields can be labeled and they support polymorphism. For example, the previous example using tuples, could be represented by the following:

Listing 2.3: Records representation in Cryptol

```
type NewType = {a : [8]; b : [1];};

t : NewType;
t = {a = 15; b = 1};
```

As we can see, each position of the record is labeled with a field name. For example, `t.a` returns the value of the field `a`. Also, the records may have polymorphism in their elements.

2.2.2 Operations and Features

Like most programming languages, Cryptol supports a large a set of operators for arithmetic, boolean, equality and comparison operations. It supports comments and literate programming which is a coding style that emphasizes documentation. Also, Cryptol is stateless and it does not support I/O, as its functions depend only on the input.

The Cryptol's operators are the following:

- **Arithmetic operators**: `+`,`-`,`*`,`/`, % (modulo), `**` (power) and `lg2` (logarithm base 2)

- **Polynomial arithmetic operators**: `pmult` (multiplication), `pdiv` (division), and `pmod` (modulus over polynomials)
- **Boolean operators**: `&` (and), `|` (or), `^` (exclusive-or), `~` (complement) and `negate` (two's complement)
- **Equality and comparison operators**: == (equality), != (non-equality), < (less than), > (greater than), <= (greater than or equal to), >= (less than or equal to)

As a functional programming language, Cryptol does not supports the *while* or *for* control structures found in most imperative languages. Instead, it uses function composition, recursion and sequence comprehension to achieve iteration. Sequence comprehension is similar to set comprehension and is useful to combine elements from distinct sequences. The *if-then-else* control structure is supported by Cryptol, allowing one to test conditional expressions and execute specific instructions based on the test result (true or false). However, as a strongly typed language, the returning type of the *then* clause must be the same as the *else* clause [cry08].

Polynomial representation and operators allow one to define and execute the basic operations of multiplication, division or modular reduction over them. To define a polynomial, it should be enclosed by `<|` and `|>`. For instance, polynomial $p(x) = x^2 + 1$ can be defined in Cryptol as `p = <| x^2 + 1|>;`.

Bit-wise operations such as shifting and rotating ones, have a strong impact in cryptographic algorithms. In Cryptol, left-shifting may be done by using the << operator and right-shifting may by using the >> operator. Left-rotate may be done by using the <<< operator and right-rotate by using the >>> operator.

As sequences are a relevant part of the Cryptol language, there are some built-in functions to manipulate them. Concatenation (`#`) or selection (`@`) are examples of these functions. Other built-in function for selection is (`@@`):

Listing 2.4: `@@` operator in Cryptol

```
1 Cryptol> [0x01 0x02 0x03] @@ [0 2]
2 [0x1 0x3]
```

In this example, it is selected from the first sequence, the values on the indexes

provided on the second sequence, passed as argument. There are other built-in functions in Cryptol, useful for sequence manipulation, as we can see in the Table 2.1.

Function	Signature	Description
!	`{a b c} (fin a) => ([a]b,[c]) -> b`	selects (from the end) the element of the index passed as argument
!!	`{a b c d} (fin a) => ([a]b,[c][d]) -> [c]b`	selects (from the end) the elements of the indexes passed as arguments
drop	`{a b c} (fin a,a >= 0) => (a,[a+b]c) -> [b]c`	returns a new sequence only leaving out the first n elements of the argument sequence
groupBy	`{a b c} (b,[a*b]c) -> [a][b]c`	groups a set of splitted sequences with the same width
join	`{a b c} [a][b]c -> [a*b]c`	concatenate the elements of a sequence
reverse	`{a b} (fin a) => [a]b -> [a]b`	reverses the elements of a sequence
split	`{a b c} [a*b]c -> [a][b]c`	splits a sequence into smaller parts
splitBy	`{a b c} (a,[a*b]c) -> [a][b]c`	splits a sequence into n parts
tail	`{a b} [a+1]b -> [a]b`	drop the first element of a sequence
take	`{a b c} (fin a,b >= 0) => (a,[a+b]c) -> [a]c`	returns a new sequence only leaving out the last n elements of the argument sequence
transpose	`{a b c} [a][b]c -> [b][a]c`	transpose the top two levels of a nested sequence
width	`{a b c} (c >= width a) => [a]b -> [c]`	returns the number of elements of the sequence passed as argument

Table 2.1: Cryptol built-in functions for sequence manipulation

In short, sequences are the natural way to structure data in Cryptol and may be used to create nested sequences of data, for instance, to represent matrices. These functions allow developers to manipulate and structure sequences and

nested sequences efficiently.

2.2.3 Compilation and Verification

The Cryptol toolkit provides back ends and evaluation modes to re-target high-level specifications to other languages, as well as verification tools. The back ends for re-targeting Cryptol code into other programming languages are C, as well as other evaluation modes, focused on hardware platforms, such as bit, LLSPIR, VHDL and FPGA [Inc08] [cry08].

The C back end produces optimized executable code. During the compilation, a set of verifications like the out-of-bounds-check are executed. This verification aims to assure that every access, on an array, founds an element. However, the C compiler has a limitation, as it can only compile Cryptol code with monomorphic functions [Inc08].

The first mode, for hardware platforms, is the bit, which evaluates if data is modelled as sequences of bits. The second evaluation mode produces as output a `.dot` file of the LLSPIR (Low Level Signal Processing Intermediate Representation). The third evaluation mode compiles the high-level specification to LLSPIR and then translates it to VHDL. VHDL was originally designed to support the whole hardware design process from the specification of the highest abstraction level to the physical level [EKP98]. It is a strongly typed language, in which every object has an associated type, specifying which type of values may be stored and which operations may be performed on it. The fourth evaluation mode, FPGA, compiles the high-level specification to LLSPIR, translates it to VHDL and then use external tools to synthesize the generated VHDL into a specific netlist. There are three modes of operation: FSIM, TSIM, FPGA_Board [Inc08].

The Cryptol tool kit also may be used to verify foreign VHDL implementations. These tools provide equivalence checking between VHDL code generated via Cryptol and given external VHDL code. With this tool, developers may import external VHDL code for a specific cryptographic algorithm and verify if it is equivalent to the VHDL generated via the Cryptol specification [Inc08].

QuickCheck is also supported by Cryptol, and can be used to generate test vectors.

2.3 CAO

Proposed by the Cryptography and Information Security Group of the University of Bristol, CAO is a cryptography-aware domain-specific language. It was taken over by the CACE[2] project, which aimed to develop a set of tools to the specification and design of applications in the cryptographic domain. In these tools, are included the CALF compiler (an extension to the CAO language), or the CAO Deductive Verification Tool, which aims to verify programs written in CAO automatically [BAC+09].

CAO is an imperative language and its syntax looks similar to C's. It was designed to quickly incorporate data types that allow an easy transcription of cryptographic primitives, based on the published specifications [BAC+09]. With the CAO language and the associated tool kit, it is expected that cryptographic developers may produce components with a higher quality and that their productivity is increased [BAC+09].

2.3.1 Type system

The CAO programming language uses variables to represent data, which may be manipulated by a set of language's operators. These variables are identified by a variable name and a type. CAO's type system is monomorphic, i.e., every constant, variable or function result must have its type declared [WFH90]. The CAO's type system is composed by primitive types and synonym types: the former represent the built-in types and the latter represent new types, defined by developers.

As we can see in the Table 2.2, primitive types are composed by the types *void*, *int* (integer with arbitrary precision) and *bool* (boolean type). It supports bit strings with an arbitrary length which may be *signed* or *unsigned* and finite fields over a body with *m* elements. Also, it supports container types such as vectors or matrices, which can store values of the same type.

Other types may be defined by using `typedef`, allowing developers to create synonym types of an existing one. Furthermore, structures similar to C's may be defined. Considering a geometric point with two coordinates (x,y), we can define

[2] http://www.cace-project.eu/

Type	Description
void	Empty type
int	Integers in Z
bool	Boolean values (True or False)
signed bits[n]	Bitstrings with length n, with signal bit
unsigned bits[n]	Bitstrings with length n, without signal bit
mod m	Finite field defined over m
vector[n] of a	Vector with n elements of the type a
matrix[n,m] of a	Matrix with n $\times$ m elements of the type a

Table 2.2: CAO's types

a new type named `Point` by using the reserved word `struct`:

Listing 2.5: A structure in CAO representing a 2D point

```
1  typedef Point := struct [
2              def x : int;
3              def y : int;
4              ];
```

Type casts or type conversions, i.e. the implicit or explicit conversion of a variable data type to another, is possible in CAO. This allows developers to convert, for instance, an integer to an unsigned bit string, as long as the unsigned bit variable has the required bits to represent the original integer. Particularly, the conversion of a vector or matrix is only possible if every member's data type is convertible.

2.3.2 Operations and Language features

Just like most general programming languages, CAO supports arithmetic, boolean, equality and comparison operations, to manipulate its data types. The supported operators are the following:

- **Arithmetic operators**: +,-,*,/, % (modulo), ** (power)
- **Boolean operators**: && (and), || (or), ^ (exclusive-or)
- **Equality and comparison operators**: == (equality), != (non-equality), <, >, <=, >=

- **Conditional expressions**: if-then-else, if-then
- **Shifts and rotates**: << (shift left), >> (shift right), <| (rotate left), |> (rotate right)
- **Sequence operators**: @ (concatenation)

It supports control structures such as `if-then-else`, used to test conditionals. However, the `else` clause may be omitted, i.e. a conditional statement may be executed exclusively with the `then` clause. For iteration, CAO provides the `while` command, which executes a block of instructions, if the boolean condition is satisfied. Two other cycle iterators are available: `seq` and `seq by`, which iterate from a lower bound to an upper bound. For the *seq by* instruction, a third value must be specified, known as the *leap*, which defines the leap that is made in the following iteration. The *seq* instruction may be seen as a *seq by* with *leap = 1*. The values used as the lower bound, upper bound and the leap must be known at the compile time. Considering the following example:

Listing 2.6: 'Using *seq* in CAO'

```
1 seq x:=0 to 15 by 4 {
2   i := i+1;
3 }
```

This `seq` iterates from 0 to 15, and since the *leap* is defined as 4, it executes 4 times.

To access an element in the position *i* of a vector, bit string or matrix *v* is done by accessing `v[i]`. Also, multiple values may be selected, using ranges like `v[i..j]`, which return a vector with the elements in the vector *v* from the position *i* to the position *j*.

Functions and procedures are similar to C's: the function name, arguments and corresponding types and the returning type need to be specified. Particularly, functions' arguments are passed by value and not by reference as happens in C. Also, in CAO a function may return multiple values.

There are three types of functions in CAO: pure functions, read-only functions and procedures [BAC+09]. The first ones do not depend on global variables and only call other pure functions, in order to guarantee that a pure function always

return the same values if called with the same arguments. Read-only functions may read global values but cannot change them and may call pure and read-only functions. The procedures may read and write into global variables and call any type of functions [BAC+09]. The distinction of types of functions do not have any implication in the implementation of programs in CAO, however it was useful in the language design. [BAC+09].

2.3.3 The CAO tool kit

The CACE project proposed a set of tools on different levels of abstraction. It had as main goals to develop tools that allow the translation from natural specifications, automatic security awareness and automatic optimization for a diversity of platforms. In this context, CAO was used to low-level side channel resistant implementation of cryptographic algorithms. Libraries for secure communication or the formal verification of developed code were also approached [BAC+09].

A set of tools, that interact directly with CAO, was developed: the CALF compiler, the CAO interpreter, the CAO VCGen (Verification Condition Generator) which uses the CAO-SL annotation language and finally, the CAO2C compiler [BAC+09].

CALF Language The CALF language was designed to be an enhanced version of the CAO language, including higher-level features that aim to provide a more user-friendly programming environment, reduce probability of programming errors, easier code re-usability, portability and manageability [BAC+09]. It implements features such as higher-order operators, dependent types and allow the implementation of polymorphic functions.

For example, consider the following definition of the parameters (a key-pair) of the DSA algorithm, a digital signature algorithm:

Listing 2.7: 'Polymorphism using CALF'

```
1 typedef DSAParams<(p : int, q : int)> := struct [
2   def g  : mod[p];
3 ];
```

This represents an abstract representation of the DSA parameters, that can be instantiated for different values of p and q. However, as all values in CAO must be known at compile time, the CALF compiler parses the developed code to generate the target CAO implementation, with concrete values for p and q.

CAO Interpreter The CAO interpreter was designed based on the formal specification of the CAO language and is an interactive interpreter of CAO programs [BAC+09]. It was designed to parse a CAO program according to the syntactic rules of the language specification but also to execute type checking against the expressions in the code.

CAO-SL The CAO-SL is a specification language, that can be used to add annotations to CAO programs. This annotation language includes the definition of function contracts with pre-conditions, post-conditions and statement annotations such as assertions, loop variants and invariants.

CAO Deductive Verification Tool This tool was designed to generate a set of proof obligations that can be discharged using a large set of automatic theorem provers such as Simplify, Alt-Ergo, Coq or Z3. It was built over existing tools such as Jessie (a plug-in for the Frama-C tool that allows static analysis of C programs) and Why, a Verification Condition Generator (VCGen). The Jessie plug-in translates the annotated CAO code to the input required by the Why tool. Then, such input is processed by one of the previously mentioned automatic theorem provers to verify if the CAO program satisfies the specified conditions.

2.4 Summary

In this chapter we have presented the definition of a DSL, focusing on the particular domain of cryptography. A brief overview on the cryptographic domain, its main primitives and the main requirements that a DSL for cryptography must satisfy, was presented. Two DSLs for cryptography were also presented: Cryptol and CAO: the former is a functional language and the latter is an imperative language. However, both were designed with a formal methods-based approach

and the main goal is that one can implement cryptographic algorithms in a clear way, assuring that the developed code is secure.

We have also shown the tools that are provided with both languages. Cryptol is composed by compilation back ends, to compile produced high-level specifications to other platforms such as C, VHDL or FPGA boards. The Cryptol language also provides verification tools to test the produced code. On the other hand, the CAO toolkit is composed by an extension to the CAO language, the CALF compiler, that aims to introduce some higher-level features like polymorphism. Also there is the CAO-SL tool that can be used to define function contracts with pre-conditions and post-conditions and the CAO Deductive Verification tool, which can be used to verify cryptographic code, developed with CAO.

In the next chapter, we will present the AES algorithm and will compare its implementation both in CAO and Cryptol. Key aspects of the implementation of the algorithm in both languages will be discussed.

Chapter 3

DSL comparison using the AES algorithm

One of the main purposes of the two presented DSLs for cryptography is to retarget a published algorithm from its formal specification to a concrete implementation in a simple and direct way. At first glance, both languages allow one to implement a published cryptographic algorithm based on its formal specification. They provide language's features to implement these algorithms such as their type systems with appropriate data types for the cryptographic domain, cycle iterators, functions or procedures or bit-wise operators. However, these two languages have distinct programming paradigms, which makes the implementation approach slightly different. One of many key aspects is iteration building: in CAO it may be implemented using the `seq` control structure; in Cryptol should be used sequence comprehensions, for instance.

In this chapter, we will address the AES algorithm, one of the most widely used cryptographic algorithms for symmetric encryption [AES01]. This algorithm uses a large set of mathematical structures and functions in its formal specification. It uses bit-wise operations such as bit-shifting and takes advantage of sophisticated mathematical structures such as finite fields and its representation using polynomials. This algorithm will be used to analyse and to compare a functional implementation written in Cryptol and one written in CAO.

Initially, the AES encryption scheme will be presented, with special focus on its cipher. Following this, the two implementations will be analysed and compared, focusing on key aspects for program development such as iteration building, func-

tion composition, bit-wise operations and polynomial implementation among other considerations.

3.1 Advanced Encryption Standard

Published in 2001 by Joan Daemen and Vincent Rijmen, Rijndael is an iterated block cipher that can be used to achieve information confidentiality [AES01]. It was later established as a standard for symmetric encryption as the Advanced Encryption Standard, replacing its predecessor DES, in many practical scenarios.

The AES encryption scheme is based on a substitution permutation network, and it was designed to be implemented both in hardware and software. It has a fixed block size of 128 bits and a variable key size of 128, 192 or 256 bits [VOMV96]. The key size also depends on the desired security level and defines the most common ciphers of the AES algorithm: AES-128, AES-192 and AES-256 [LLS09]. Additionally, the key size also defines the number of rounds (**Nr**) to be performed in an AES run, as we can see in the Figure 3.1.

	Key Length	**Block Size**	**Number of Rounds**
AES-128	4	4	10
AES-192	6	4	12
AES-256	8	4	14

Figure 3.1: Relationship between key length and the number of rounds

The algorithm is divided into two major steps: the Key Expansion and the Cipher (and Inverse Cipher) steps. The Key Expansion step takes the original cipher key, expanding it to **Nr+1** separate keys. The Cipher step iterates through **Nr** rounds, with an auxiliary 4×4 matrix of bytes, denominated as the *state* [DR99].

3.1.1 Cipher

Initially, the input text is copied to the state array, as illustrated in the Figure 3.2. After iterating through **Nr** rounds, the cipher's output is the final state of these iterations. The process for the Inverse Cipher is similar [DR99]: it takes the cipher text as input and retuns the plain text as output.

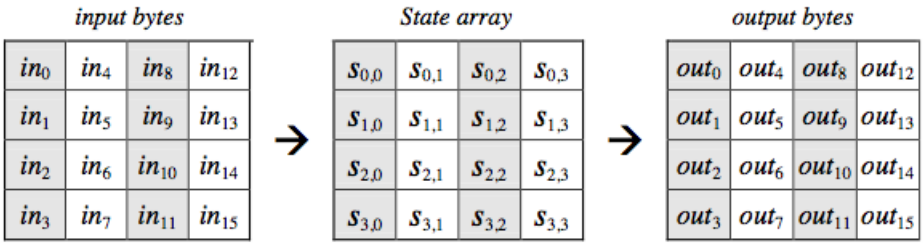

Figure 3.2: Cipher process

Each round consists of the sequence of four byte-oriented transformations: SubBytes, ShiftRows, MixColumns and AddRoundKey [DR99]:

1. **SubBytes** byte substitution using a S-box (substitution box).
2. **ShiftRows** shifting rows of the state array by different offsets.
3. **MixColumns** mixing the data within each column of the State matrix.
4. **AddRoundKey** adds a round key to the state.

Additionally, the first round is preceded by an additional AddRoundKey and the final round does not apply the MixColumns tranformation [DR99]. Generically, the AES cipher's looks like the Figure 3.3.

AddRoundKey $\Rightarrow$

(SubBytes $\Rightarrow$ ShiftRow $\Rightarrow$ MixColumns $\Rightarrow$ AddRoundKey $\Rightarrow$) $\times 9$

SubBytes $\Rightarrow$ ShiftRow $\Rightarrow$ AddRoundKey

Figure 3.3: AES-128 encryption scheme with 10 rounds

SubBytes The SubBytes transformation operates on each byte of the state, introducing non-linearity into the AES cipher [AES01]. It consists on a direct substitution of each input byte, by looking up on a substitution table (S-box). For the inverse cipher, there is also an inverse substitution table. The Rijndael S-box used in the AES cipher looks like the table in the Figure 3.4.

Once built, the substitution process takes on each input byte of the state and returns the corresponding value, by looking up on the S-box table. For example,

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**a**	**b**	**c**	**d**	**e**	**f**
0	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
1	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
2	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
3	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
4	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
5	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
6	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
7	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
8	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
9	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
a	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
b	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
c	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
d	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
e	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
f	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

Figure 3.4: Substitution-box table for the AES algorithm

considering {62} as input for the S-box, the output consists on verifying the intersection between the row with index {6} and the column with the index {2}, which as we can see in the Figure 3.4 outputs {aa} [AES01].

However, the building process of the S-box table is complex, involving two steps:

- Take the multiplicative inverse in GF(2^8)

- Apply the affine transformation over GF(2)

The multiplicative inverse in GF(2^8) of a polynomial $a(x)$ corresponds to $a(x)^{-1}$ modulo an irreducible polynomial, which in the case of AES is $i(x) = x^8 + x^4 + x^3 + x + 1$ [DR99]. The affine transformation may be represented by the operations represented in the Figure 3.5.

$$\begin{bmatrix} b'_0 \\ b'_1 \\ b'_2 \\ b'_3 \\ b'_4 \\ b'_5 \\ b'_6 \\ b'_7 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \\ b_4 \\ b_5 \\ b_6 \\ b_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

Figure 3.5: Affine transformation used in the S-box construction

ShiftRows The ShiftRows transformation shifts the last three rows of the state over different offsets. In the AES-128 cipher, with **Nb** = 4, the following shifts occurs:

- First row is not shifted
- Second row is shifted with offset = 1
- Third row is shifted with offset = 2
- Fourth row is shifted with offset = 3

MixColumns The MixColumns transformation operates column-by-column on the state, considering each column as a polynomial over GF(2^8). Each of these polynomials are multiplied by $a(x) = 3x^3 + 3x^2 + x + 2$ modulo $x^4 + 1$. The output state of this transformation may be seen as $s'(x) = a(x) \otimes s(x)$, where $s(x)$ and $s'(x)$ represent the input and the output states, respectively. As explained in [DR02], this operation can be seen as a matrix multiplication:

$$\begin{bmatrix} s'_0, c \\ s'_1, c \\ s'_2, c \\ s'_3, c \end{bmatrix} = \begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \otimes \begin{bmatrix} s_0, c \\ s_1, c \\ s_2, c \\ s_3, c \end{bmatrix}$$

Figure 3.6: MixColumns transformation

AddRoundKey The AddRoundKey transformation applies the bit-wise exclusive-or operation to each byte of the round key and each byte of the state: $s'(x) =$

$s(x) \oplus k_i$, where k_i, $s(x)$ and $s'(x)$ represent the key of the round i, the input state and the output state, respectively.

3.2 AES Implementations

As seen in the Chapter 2, Cryptol is a functional language based on function composition, recursion and sequence comprehension, representing every type of data via sequences of bits. On the other hand, the CAO language is an imperative one, similar to C, and uses the most common control structures of the paradigm such as while loops or if-then-else conditional statements. In this section, two distinct implementations of the same AES algorithm, will be compared based on the following points of analysis: which types can be used to store the state of the AES algorithm, the construction of the S-box, iteration through indexed structures of data such as sequences or vectors/matrices, polynomials and bit-wise operations.

3.2.1 Types and the State

The CAO language provides a set of types that can be useful to represent data, including container types, such as vectors or matrices that can be used to structure and store data [BAC+09]. On the other hand, the Cryptol type system is composed by the Bit type, which can be used to define new types based on sequences [cry08].

As seen previously, the AES-128 state is composed by a 4×4 matrix of bytes (8 bits). One way to represent this in Cryptol is by using a `[4*4][8]` state or `[4][4][8]`, representing that matrix.

In the CAO's implementation may be used the container type `matrix`, to represent the 4×4 matrix of bytes. This can be used to represent either the state (`S`) and the round key (`K`):

The `GF2` type represents the finite field GF(2). The `GF2N` type is an extension of `GF2`, representing the finite field GF(2^8).

Listing 3.1: Types for the AES State in CAO

```
typedef S := matrix[4,4] of GF2N;
typedef K := matrix[4,4] of GF2N;

typedef GF2 := mod[ 2 ];
typedef GF2N := mod[ GF2<X> / X**8 + X**4 + X**3 + X + 1 ];
```

3.2.2 Function Composition

As in most general purpose languages, composing functions by using the output of a function as input of another is largely used by developers. In the Round function, the SubBytes transformation takes the state as input. Then, after the ShiftRows, MixColumn and AddRoundKeys, the state is returned as output.

These four transformations are composed in order to execute one round of the AES algorithm. The implementation of the `Round` function in Cryptol looks like the following:

Listing 3.2: AES Round implementation in Cryptol

```
Round : ([4][4][8], [4][4][8]) -> [4][4][8];
Round (State, RoundKey) = State3 ^ RoundKey
  where {
    State1 = ByteSub State;
    State2 = ShiftRow State1;
    State3 = MixColumn State2;
  };
```

It stores temporary states (State1, State2 and State3) as the intermediate results from each transformation, returning the output of the MixColumns transformation XORed with the RoundKey.

On the other hand, the CAO's approach also focuses on composing the functions:

Listing 3.3: AES Round implementation in CAO

```
def FullRound( s : S, k : K ) : S
{
  return MixColumns( ShiftRows( SubBytes(s) ) ) + k;
}
```

Both functions take the state and the round key given as input and return the exclusive-or of the output of the MixColumns transformation with the round key. Note that the `K` type is also defined as a `GF2N` and so, the exclusive-or operation is done by the operator `+`.

3.2.3 Iteration

Iteration through container types such as vectors or matrices in CAO or sequences in Cryptol is a key point for the implementation of cryptographic algorithms. As the programming paradigms differ, it is important to identify how it could be implemented in these languages.

The AES cipher iterates through **Nr** rounds: the Round function is called **Nr** times. Note that there is an additional AddRoundKey before the **Nr** rounds and that MixColumns is not applied in the last round.

In Cryptol, the approach to implement this function is through the use of recursive sequence comprehension lists. The following function implements the `Rounds` function, according to the AES standard:

Listing 3.4: AES Rounds implementation in Cryptol

```
1 Rounds (State, (initialKey, rndKeys, finalKey)) = final
2   where {
3       istate = State ^ initialKey;
4       rnds = [istate] # [| Round (state, key)
5                          || state <- rnds
6                          || key <- rndKeys |];
7       final = FinalRound (last rnds, finalKey);
8   };
```

After executing an additional AddRoundKey on the initial state, this function iterates through **Nr** rounds. This is done by the use of a sequence comprehension lists, that take the current state and a key from `rndKeys` and calls the Round function. Finally, the `FinalRound`, which is a particular implementation of the round without the MixColumns transformation, is executed.

The CAO language supports while loops or iteration through the use of the *seq* iterator. Considering that the round keys are stored in the vector `keys`, the following code snippet may be used to implement the rounds of the AES algorithm:

Listing 3.5: Code snippet of the AES Rounds implementation in CAO

```
1   // (...)
2   seq i := 1 to 9 {
3   r := FullRound( r,keys[i] );
4   }
5
6   return ShiftRows( SubBytes(r) ) + keys[10];
7 }
```

The `FullRound` function is called nine times, passing as argument the current state and the round key. In the return statement, is executed the last round of the AES cipher, executing only the SubBytes, ShiftRows and the final AddRoundKey.

3.2.4 Indexed structures manipulation

The natural way to structure data in Cryptol is via sequences, which allow one to group bits as intended. CAO's containers such as vectors or matrices may be used for the same purpose. However, after store data into these structures, it is necessary to manipulate them.

Cryptol provides a large set of pre-defined functions to manipulate sequences, such as `join` and `split`, as seen in the Table 2.1 on the page 12. These functions, are polymorphic and could be used with any sequence size. On the other hand CAO does not have these built-in functions and they must be written specifically for each scenario.

This is related to the polymorphism that the Cryptol's type system supports unlike CAO, that only supports polymorphism via the CALF Extension [BAC+09].

3.2.5 The S-box construction

As seen previously in the formal specification of the Rijndael S-box [DR99], it is built in two steps:

- Take the multiplicative inverse in GF(2^8)
- Apply the affine transformation (over GF(2))

Both implementations in Cryptol and CAO have similar approaches to implement this component of the AES algorithm. In the Cryptol implementation, the S-box table results precisely from the application of the affine function to the inverse of every element $0 \leq x < 256$:

Listing 3.6: Rijndael S-box function in Cryptol

```
1 sbox : [256][8];
2 sbox = [| affine (inverse x) || x <- [0 .. 255] |];
3
4 affine : [8] -> [8];
5 affine xs = join (mmultBit (affMat, split xs)) ^ 0x63;
```

The affine transformation implementation in Cryptol executes the required operations to execute the matrix multiplication represented in the Figure 3.5, however, it will not be explained here in detail for the sake of clarity. Even so, it is important to notice the exclusive-or with the number `0x63`, which represents the vector $[01100011]$.

The CAO implementation also intends to reproduce the affine transformation represented in Figure 3.5 on the page 25. It represents explicitly the matrix `M` and the vector `C` as we can see in the following code snippet:

Listing 3.7: Auxiliary matrix and vector to the affine transformation in CAO

```
1  def M : matrix[8,8] of GF2 := {  1, 0, 0, 0, 1, 1, 1, 1,
2                                   1, 1, 0, 0, 0, 1, 1, 1,
3                                   1, 1, 1, 0, 0, 0, 1, 1,
4                                   1, 1, 1, 1, 0, 0, 0, 1,
5                                   1, 1, 1, 1, 1, 0, 0, 0,
6                                   0, 1, 1, 1, 1, 1, 0, 0,
7                                   0, 0, 1, 1, 1, 1, 1, 0,
8                                   0, 0, 0, 1, 1, 1, 1, 1 };
9
10 def C : vector[8] of GF2 := { 1, 1, 0, 0, 0, 1, 1, 0 };
```

Then, the S-box function in CAO boils down to the application of the inverse transformation over GF(2^8) followed by the affine transformation:

The `M` matrix is multiplied by the result of the inverse of the input `e`, then the affine transformation is applied by executing the sum operation with the vector `C`. To note that some casts between `GF2N` and `GF2V` need to be applied, in order to execute

Listing 3.8: Rijndael S-box function in CAO

```
1 def SBox( e : GF2N ) : GF2N
2 {
3   return (GF2N)( (GF2N)( M * (GF2V)( 1 / e ) ) + (GF2N)C );
4 }
```

the matrix multiplication.

3.2.6 Polynomial representation

Polynomials play a big role in the cryptographic domain. They are used in a large set of cryptographic primitives, and the AES algorithm is no exception. AES uses polynomials in the key expansion step and in the MixColumns round transformation [DR99]. As operations over finite fields such as GF(2^8) may be also represented as polynomials [IEE99], CAO's native type allows developers to directly represent them. On the other hand, Cryptol does not have native support for finite fields, supporting only polynomials with binary coefficients.

Considering the expansion key step, in which the the irreducible polynomial of degree 8 ($i(x) = x^8 + x^4 + x^3 + x + 1$) is used. This polynomial may be represented in Cryptol as:

Listing 3.9: Representation of $x^8 + x^4 + x^3 + x + 1$ in Cryptol

```
1 irred : [9];
2 irred = <|x^8 + x^4 + x^3 + x + 1|>;
```

The `irred` polynomial requires at least 9 bits to be represented. As every variable is represented using bits in Cryptol, that polynomial is similar to `0b100011011`, which represents its coefficients.

CAO's representation of polynomials is made based on finite fields. For instance, the irreducible polynomial $i(x)$ may be represented as:

Concretly, the new type `GF2N`, defines a finite field (used in the state), in which is performed for every operation over it the modular reduction over the polynomial $i(x)$.

Listing 3.10: Representation of $x^8 + x^4 + x^3 + x + 1$ in CAO

```
typedef GF2 := mod[ 2 ];
typedef GF2N := mod[ GF2<X> / X**8 + X**4 + X**3 + X + 1 ];
```

3.2.7 Bit-wise operations

Most cryptographic algorithms use bit strings to represent data, making bit-wise operations a very relevant feature in DSLs for cryptography. As seen in the Chapter 2, both of the two presented languages support shifts, rotates or permutations. These operations are present in a large set of cryptographic algorithms, like the AES ShiftRows operation.

In the Cryptol implementation, the input state is divided into the four rows, and then each row is rotated by an offset *i*:

Listing 3.11: ShiftRows transformation in Cryptol

```
ShiftRow : [4][4][8] -> [4][4][8];
ShiftRow state =
        [| row <<< i
        || row <- state
        || i <- [ 0 .. 3 ] |];
```

The CAO's implementation is similar, also taking the state as input and rotating every row by a different offset *i*:

Listing 3.12: ShiftRows transformation in CAO

```
def ShiftRows( s : S) : S
{
  def r : S;
  seq i := 0 to 3 {
    r[i,0..3] := (vector[4] of GF2N) s[i,0..3] |> i;
  }
  return r;
}
```

To note that in this function, it was used the selector of multiple indexes `[i,0..3]`. Furthermore, it was necessary to cast the output of the rotation to a `vector[4] of GF2N`, which is the type of each row in the state `S`.

3.3 Summary

In this chapter, we have presented the AES algorithm and how it works. Particularly, the encryption scheme was focused, leaving behind the key expansion step and the inverse cipher.

Followed by this, we have compared the two DSLs introduced in the Chapter 2. The type systems of the two languages were focused: the Cryptol type system is composed exclusively by the bit type and sequences built from them; on the other hand, CAO provides basic types such as integers and booleans, as well as container types. Also, the polymorphism that Cryptol provides is also useful when manipulating sequences. CAO's monomorphic type system, forces the programmer to specify functions to manipulate vectors. Other features such as function composition, iteration, polynomials and bit-wise operations were also approached.

At first glance, both languages allow one to implement a cryptographic algorithm from its formal specification. However, it was clear that the paradigms' characteristics make the implementation of an algorithm distinct in both languages. Cryptol focuses on sequence comprehension lists, function composition and recursion. On the other side, CAO as an imperative language aim to be stateful and take advantage of control structures such as `while`'s and `seq`'s.

Chapter 4

Compilation to hardware

The design of hardware components, in particular those that use cryptographic primitives, is a sensitive area, mainly because these components are incorporated into critical systems. Ensuring that the integrity of these systems is not violated and that an attacker may not obtain sensitive information, are important aspects to be considered [Huf10].

Hardware systems focused on cryptography require a different design approach when compared to software ones. Although both aim to protect sensitive information and avoid attacks from malicious agents, hardware systems are vulnerable to a different type of attacks, such as timing and side-channel ones. These attacks require additional attention and must be considered in early design phases [Huf10]. Also, the design of cryptographic components for hardware platforms require an additional knowledge of the target platform. It is necessary to know how to take advantage from each component of the hardware and essentialy, to know how to keep the system secure.

Field-programmable Gate Array (FPGAs) are a customizable and low-cost solution to prototype hardware components [Huf10]. It supports most bit-level operations to implement bit shifting or permutations, which are required by block ciphers, public key cryptography or other primitives, making it a first choice to design hardware components [Huf10].

In this chapter we will address the compilation of high-level specifications written in Cryptol to hardware platforms, focusing on the VHDL and FPGA back ends of the Cryptol toolkit. The limitations of these back ends will be analysed with the

example of the previous chapter: the AES algorithm. Additionally, transformations described detailed in [Inc08] will be addressed.

Later on, we will transform the implemented algorithm into a 32-bit version, replacing the round transformations into a simple set of lookups on T-boxes tables, based on the AES specification [DR99]. All these transformations are based on the original AES, using the Rijndael S-box.

4.1 Cryptol's Back Ends Limitations

The Cryptol tool kit provides a set of compilation tools that allows one to re-target high-level specifications to other languages, such as C, SPIR, VHDL or FPGA Board [Inc08]. However, these compilation back ends do not support the full set of features provided by the Cryptol language. Some language's elements may not be used in the implementations of high-level specifications to be translated to hardware platforms such as VHDL of FPGA [Inc08]. Also, some language's features may complicate the generation of efficient and optimized circuits [Inc08].

Most of the limitations of the Cryptol's back ends are related to the targeted platforms. For example, the VHDL/FPGA Board back ends have the following limitations:

- the VHDL/FPGA compiler only supports division by powers of 2
- the VHDL/FPGA compiler does not support primitive recursion
- the VHDL/FPGA compiler only partially supports higher-order functions

The C back end does not support polymorphic functions [Inc08]. Also, all values must be known at compile time.

Therefore, the implementation of a high-level specification to be re-targeted to one of these back ends, it is necessary to remove recursion, do not use higher-order functions, and polymorphic functions.

4.2 Compiling Regular AES Implementation

The AES algorithm uses a large set of mathematical primitives and bit-wise operations, as seen in the Chapter 3. However, the AES implementation presented there uses higher-order constructs such as polymorphic functions, making the translation to VHDL or FPGA not possible.

For instance, after loading the AES implementation in the Cryptol to the Cryptol's interpreter, and after setting it to the VHDL translation mode (`:set vhdl`), the following error message appears:

```
1  Sorry, not implemented: unsupported form of recursion.
2  Currently we only allow recursive value bindings.
3  The offending binding:
4  rec find1 (xs_find1,i_find1) : ([256][8],[8]) -> [8] =
5          if
6          xs_find1 @{256,[8],8} i_find1 =={[8]} 1 {8}
7          then
8          i_find1
9          else
10         find1 (xs_find1, i_find1 +{8,Bit} 1 {8})
```

This error is related to the use of recursion in the function `find1`, used to calculate the Rijndael S-box, in particularly the inverse step.

In fact, there are some steps that must be followed before the AES algorithm implementation may be translated to VHDL: the Rijndael S-box will be explicitly defined rather than calculated, the MixColumns transformation will be redesigned in order to do not use recursion, and the `Rounds` and `Round` functions will be defined as monomorphic. This steps are essential to eliminate the higher-order constructs that are not supported by the back ends of Cryptol.

4.2.1 SubBytes

The SubBytes transformation maps every byte of the input state into the output state by executing a substitution using the Rijndael S-box [DR99]. In Cryptol, this

can be represented by the following functions:

Listing 4.1: ByteSub transformation

```
ByteSub : [4][4][8] -> [4][4][8];
ByteSub state =
  [| [| Sbox x || x <- row |] || row <- state |];

Sbox : [8] -> [8];
Sbox x = sbox @ x;
```

In fact, the SubBytes function uses the `sbox` function, which return the output vale for each byte of the input. This function may be defined, according to the formal specification, by calculating the multiplicative inverse over GF(2^8) and by applying the affine transformation over GF(2). An alternative, is to explicitly define the S-box presented in the Figure 3.4 in Cryptol. This can be seen in the Listing 4.2.

Listing 4.2: sbox function explicitly defined in Cryptol

```
sbox : [256][8];
sbox =
   [
   0x63 0x7c 0x77 0x7b 0xf2 0x6b 0x6f 0xc5 0x30 0x01 0x67 0x2b 0xfe 0xd7 0xab 0x76
   0xca 0x82 0xc9 0x7d 0xfa 0x59 0x47 0xf0 0xad 0xd4 0xa2 0xaf 0x9c 0xa4 0x72 0xc0
   0xb7 0xfd 0x93 0x26 0x36 0x3f 0xf7 0xcc 0x34 0xa5 0xe5 0xf1 0x71 0xd8 0x31 0x15
   0x04 0xc7 0x23 0xc3 0x18 0x96 0x05 0x9a 0x07 0x12 0x80 0xe2 0xeb 0x27 0xb2 0x75
   0x09 0x83 0x2c 0x1a 0x1b 0x6e 0x5a 0xa0 0x52 0x3b 0xd6 0xb3 0x29 0xe3 0x2f 0x84
   0x53 0xd1 0x00 0xed 0x20 0xfc 0xb1 0x5b 0x6a 0xcb 0xbe 0x39 0x4a 0x4c 0x58 0xcf
   0xd0 0xef 0xaa 0xfb 0x43 0x4d 0x33 0x85 0x45 0xf9 0x02 0x7f 0x50 0x3c 0x9f 0xa8
   0x51 0xa3 0x40 0x8f 0x92 0x9d 0x38 0xf5 0xbc 0xb6 0xda 0x21 0x10 0xff 0xf3 0xd2
   0xcd 0x0c 0x13 0xec 0x5f 0x97 0x44 0x17 0xc4 0xa7 0x7e 0x3d 0x64 0x5d 0x19 0x73
   0x60 0x81 0x4f 0xdc 0x22 0x2a 0x90 0x88 0x46 0xee 0xb8 0x14 0xde 0x5e 0x0b 0xdb
   0xe0 0x32 0x3a 0x0a 0x49 0x06 0x24 0x5c 0xc2 0xd3 0xac 0x62 0x91 0x95 0xe4 0x79
   0xe7 0xc8 0x37 0x6d 0x8d 0xd5 0x4e 0xa9 0x6c 0x56 0xf4 0xea 0x65 0x7a 0xae 0x08
   0xba 0x78 0x25 0x2e 0x1c 0xa6 0xb4 0xc6 0xe8 0xdd 0x74 0x1f 0x4b 0xbd 0x8b 0x8a
   0x70 0x3e 0xb5 0x66 0x48 0x03 0xf6 0x0e 0x61 0x35 0x57 0xb9 0x86 0xc1 0x1d 0x9e
   0xe1 0xf8 0x98 0x11 0x69 0xd9 0x8e 0x94 0x9b 0x1e 0x87 0xe9 0xce 0x55 0x28 0xdf
   0x8c 0xa1 0x89 0x0d 0xbf 0xe6 0x42 0x68 0x41 0x99 0x2d 0x0f 0xb0 0x54 0xbb 0x16
   ];
```

As we can see, the functions `SubBytes`, `Sbox` and `sbox`, used in the SubBytes transformation, are all monomorphic. Also, all values are known at compile time, and does not use any recursive function, making the SubBytes transformation translatable to hardware platforms such as VHDL.

4.2.2 MixColumns

The MixColumns transformation considers every column of the state as polynomials over GF(2^8) and multiplies it modulo $x^4 + 1$ with the fixed polynomial $a(x) = 3x^3 + 3x^2 + x + 2$ [DR99]. The implementation in Cryptol that is being followed, implements this transformation through the use of complex polymorphic functions, essentially used for the polynomial representation.

As explained in [Inc08], the MixColumns transformation may be defined in a way that does not use polymorphic functions, replacing `gPower` and `gTimes` for monomorphic functions. Specifically, the MixColumns is defined as presented in the Listing 4.8.

Listing 4.3: MixColumn function in Cryptol

```
1 MixColumn : [4][Nb][8] −> [4][Nb][8];
2 MixColumn state =
3   transpose [| multCol (cx, col)
4              || col <− transpose state
5              |];
```

In this function, the main problem is related to `multCol`, which calls a large set of polymorphic functions. A solution, described in [Inc08], is to explicitly define two functions `gTimes2` and `gTimes3` which are instances of the function `gTimes`.

Listing 4.4: MixColumn function in Cryptol translatable to hardware

```
1  mixColumn_prime : [4][8] −> [4][8];
2  mixColumn_prime([y0 y1 y2 y3]) = [
3          (gTimes2(y0) ^ gTimes3(y1) ^ y2 ^ y3)
4          (y0 ^ gTimes2(y1) ^ gTimes3(y2) ^ y3)
5          (y0 ^ y1 ^ gTimes2(y2) ^ gTimes3(y3))
6          (gTimes3(y0) ^ y1 ^ y2 ^ gTimes2(y3))
7          ];
8
9  MixColumn_prime : [4][Nb][8] −> [4][Nb][8];
10 MixColumn_prime (state) = transpose(
11         [| mixColumn_prime column
12         || column <− transpose(state)
13         |]
14     );
```

The new function `MixColumn_prime`, calls the `mixColumn_prime` function, which is

monomorphic, allowing the compilation to VHDL.

4.2.3 The Encryption Scheme

The encryption scheme on AES is based on two steps, the key schedule and the cipher step. The cipher step, focused on this work, is based on the application of **Nr** rounds, using the following `Round`, `Rounds` and `encrypt` functions. The `Round` and `Rounds` function is similar to the one presented in Chapter 3:

Listing 4.5: Rounds function in Cryptol

```
1  Rounds : {a} ([4][4][8], ([4][4][8],[a][4][4][8],[4][4][8])) -> [4][4][8];
2  Rounds (State, (initialKey, rndKeys, finalKey))
3    = final
4    where {
5      istate = State ^ initialKey;
6      rnds = [istate] # [| Round (state, key)
7                          || state <- rnds
8                          || key <- rndKeys |];
9      final = FinalRound (last rnds, finalKey);
10   };
```

However, as we can see in the `Rounds` function signature, this function may accept *a*-sized values, in the `rndKeys` arguments, i.e. this function can be used in AES-128, AES-192 and AES-256. When attempting to translate these functions to VHDL, the Cryptol's interpreter outputs the following message:

Listing 4.6: Type of the Rounds function in Cryptol

```
1  <polymorphic value> : {a} ([4][4][8],([4][4][8],[a][4][4][8],[4][4][8])) -> [4][4][8]
```

In our scenario, we intend to compile the Rounds function focusing on the AES128 cipher, which uses 10 rounds and 11 keys: 1 initial round for the additional AddRoundKey and 10 round keys. So, we can explicitly change the function signature of the function by defining

Listing 4.7: Rounds function signature in Cryptol

```
1  Rounds : ([4][4][8], ([4][4][8],[9][4][4][8],[4][4][8])) -> [4][4][8];
```

After this transformation, the `Rounds` function is now monomorphic and it can be translatable to VHDL. To finish the encryption scheme of the AES, consider the `encrypt` function, that receives as argument a set of round keys and the plain text:

Listing 4.8: encrypt function in Cryptol

```
1 encrypt (XK, PT) = unstripe (Rounds (State, XK))
2   where {
3     State : [4][4][8];
4     State = stripe PT;
5   };
```

This function can be directly translated to VHDL, as it does not use polymorphism or recursion. The `stripe` and `unstripe` functions are used as auxiliary: the former is used to transform the plain text given as input into the state and the latter is used in the inverse process [cry08].

4.3 32-bit AES implementation

The Rijndael cipher was designed to fit any type of architecture, i.e. to be implemented on a large set of platforms, that can be hardware or software. It was intended to be implemented in a large set of processors, distinct computer architectures and dedicated hardware [DR99].

The most common implementations of the AES algorithm in hardware are designed for 8-bit and 32-bit processors. The former fits essentially in smart cards and the latter was designed to fit in personal computers [DR99]. In this section, we will present the 32-bit version of the AES algorithm.

4.3.1 Formal Specification

The four transformations of the AES round may be combined into a single operation, which consists on a set of table lookups, based on the original Rijndael S-box (Figure 3.4) [DR99]. The AES state may be seen as a 4×4 matrix of bytes which may be also represented as a vector of 4 elements with 32 bits each. The AES

32-bit version processes the state 32-bit at a time, by combining each column of the state (composed by 4 bytes) into one block.

Considering that one column of the round output is represented by *e* and the input is represented by *a*, $a_{i,j}$ represent the input byte of the row *i* and the column *j*, an AES round may be expressed as:

$$e_j = T_0\ [a_{0,j}] \oplus T_1\ [a_{1,j}] \oplus T_2\ [a_{2,j}] \oplus T_3\ [a_{3,j}] \oplus k_j$$

Each T_i, with $0 \leq i < 3$, is known as a T-box, a table composed by 256 words of 4-bytes each entry. These tables are also built based on the original Rijndael S-box, combining also the ShiftRows and MixColumns transformation:

$$T_0\,[a] = \begin{bmatrix} S[a] \bullet 02 \\ S[a] \\ S[a] \\ S[a] \bullet 03 \end{bmatrix} \quad T_1\,[a] = \begin{bmatrix} S[a] \bullet 03 \\ S[a] \bullet 02 \\ S[a] \\ S[a] \end{bmatrix}$$

$$T_2\,[a] = \begin{bmatrix} S[a] \\ S[a] \bullet 03 \\ S[a] \bullet 02 \\ S[a] \end{bmatrix} \quad T_3\,[a] = \begin{bmatrix} S[a] \\ S[a] \\ S[a] \bullet 03 \\ S[a] \bullet 02 \end{bmatrix}$$

Figure 4.1: T-boxes specification

The construction of these T-boxes is obtained by combining the four transformations of the AES round. The set of steps to execute to build these T-boxes will not be explained here, however, it is fully described in [DR99].

4.3.2 Implementation in Cryptol

As presented in the previous section, the Rijndael S-box may be used to build T-boxes, transforming the AES round into a set of lookup in these tables. To build these tables, it is necessary to implement functions to calculate the T-boxes elements, as explained in the Figure 4.1.

Considering the T_0-box, each element *a*, may be built according to the following function:

Listing 4.9: Calculation of the values for the T_0-box

```
1 T0_func : [8] −> [4][8];
2 T0_func(a) = [(gtimes2 s) s s (gtimes3 s)]
3         where s = Sbox(a);
```

This aims to represent the operations presented in Figure 4.1: the substitution of the input *a* by the corresponding value of the S-box *e* is made, and then the sequence $[s^2\ s\ s\ s^3]$ is returned. The auxiliary functions `gtimes2` and `gtimes3` calculates s^2 and s^3, respectively. As we are using 32-bit, it is necessary to execute an auxiliary function to join these `[4][8]` elements into `[32]` words. This can be done through the function `T0_table`, which returns 32-bit words.

Listing 4.10: Calculation of the T_0-box

```
1 T0_table = [| join (T0_func(a)) || a <− [0..255] |];
```

Appling the join function to $0 \leq X \leq 255$, returns a table with 256 entries with 32 bit each. The generated `T0_table` is presented in the Appendix A.

The other T-boxes may be calculated by similar functions, which only rotate the elements in the output sequence:

Listing 4.11: Calculation of the others T-boxes

```
1  T1_func : [8] −> [4][8];
2  T1_func(a) = [(gtimes3 s) (gtimes2 s) s s]
3          where s = Sbox(a);
4
5  T2_func : [8] −> [4][8];
6  T2_func(a) = [s (gtimes3 s) (gtimes2 s) s]
7          where s = Sbox(a);
8
9  T3_func : [8] −> [4][8];
10 T3_func(a) = [s s (gtimes3 s) (gtimes2 s)]
11         where s = Sbox(a);
```

With these T-boxes, it is now possible to compile the AES128 encryption scheme, using 32-bit, to VHDL although it is still necessary to update the Round function to execute the lookups on these tables.

To note in this function, the *if-then-else* condition for the final round, which unlike

Listing 4.12: Round function using T-boxes

```
1  Round : ([4][4*8], [4][4*8], Bit) -> [4][4*8];
2  Round (State, RoundKey, final)
3    = d ^ RoundKey
4    where {
5      a : [4][4][8];
6      a = [| split s || s <- State |];
7      d = [| if final
8              then (t0_32 ^ t1_32 ^ t2_32 ^ t3_32
9                      where {
10                       t0_32 = ((Sbox (a @ ((j) % 4) @ 0)) # zero);
11                       t1_32 = ((Sbox (a @ ((j+1) % 4) @ 1)) # zero) << 8;
12                       t2_32 = ((Sbox (a @ ((j+2) % 4) @ 2)) # zero) << 16;
13                       t3_32 = ((Sbox (a @ ((j+3) % 4) @ 3)) # zero) << 24;
14                     }
15                   )
16             else (t0 ^ t1 ^ t2 ^ t3
17               where {
18                 t0 = T0_table @ (a @ ((j) % 4) @ 0);
19                 t1 = T1_table @ (a @ ((j+1) % 4) @ 1);
20                 t2 = T2_table @ (a @ ((j+2) % 4) @ 2);
21                 t3 = T3_table @ (a @ ((j+3) % 4) @ 3);
22               }
23               )
24         || j <- [0 .. 3]
25         |];
26   };
```

the normal round, does not execute the lookups on the T-boxes, using the original S-box instead.

4.4 Summary

In this chapter, we approached the compilation of high-level specifications written in Cryptol to hardware platforms, focusing on the VHDL back end of the Cryptol toolkit.

There are some limitations related to the VHDL back end, that does not support the full set of Cryptol's features. Higher-order functions, recursive calls and polymorphic functions are examples of features that must not be used in high-level specifications that aim to be translated to a hardware platform such as VHDL,

afterwards.

Some transformations were applied to the AES algorithm, in order to make it translatable to VHDL. Two versions were presented: the regular version presented in the previous chapter and a 32-bit version.

Chapter 5

CAO2Cryptol: A compilation tool

The two DSLs for cryptography presented through this document, Cryptol and CAO, provide the appropriate features to implement a cryptographic algorithm, based on its formal specification. Also, the two languages have toolkits associated: the Cryptol language is complemented by a verification suite and a set of compilers that allow one to re-target Cryptol code to C or VHDL; the CAO language is complemented by the CALF compiler that introduces higher-order features in the language, the CAO-SL that allows one to specify function contracts and the CAO Deductive Verification Tool, that allows one to generate a set of proof obligations.

Unlike the Cryptol toolkit, CAO does not support the translation of high-level specifications to any type of platform. It would be desirable to have a compiler that could take on the implementation of a cryptographic algorithm in CAO, and translate it to a hardware description language, such as VHDL.

As we seen in the Chapter 4, the Cryptol back ends focused on hardware platforms (VHDL and FPGA Board), only support a subset of the language. Higher-order functions, polymorphism or primitive recursion are not supported by the VHDL or FPGA compilers. The code using these features should be rewritten, so that it can be re-targeted to these platforms.

In this chapter, we intend to analyse and prototype a compiler that takes CAO source code as input, and produces Cryptol source code as output: the CAO2Cryptol compiler. Particularly, it is expected that the generated code may be translatable to hardware, i.e. that the Cryptol code does not use recursion, polymorphism or

higher-order functions.

The two languages were already explained in detail in the Chapter 2, followed by a practical example, the AES in the Chapter 3, providing the required background to analyse the requirements of the CAO2Cryptol tool: the limitations of the Cryptol back ends, its type systems and the characteristics of each language.

5.1 Requirements Analysis

In order to design the CAO2Cryptol, it is necessary to understand and define which features of CAO can be supported for compilation and in what elements of the target language, they will match. Also, since it is intended that the generated source code may be translated to hardware description languages, it is important to consider the limitations of the Cryptol language, as presented in Chapter 4.

5.1.1 Languages Analysis

Although both Cryptol and CAO are domain-specific languages for cryptography, they are distinct in a lot of features of the languages. Actually, they are based on two distinct programming paradigms, which makes the approaches used to build high-level specification of cryptographic algorithms different.

The Cryptol language is a functional language and focuses itself in lambda calculus and function application. The Haskell programming language, for instance, is a pure functional programming language which uses a static type system. However, it allows side effects through the use of monads. The Cryptol language, is stateless and its functions does not produce side effects.

The CAO programming language, is based on the imperative paradigm, and each program is composed by a list of statements, where each one may change the state of the program.

Type systems

As presented in the Chapter 2, the CAO's type system is composed by the following types: `void`, `int`, `bool`, `signed bits`, `unsigned bits`, finite fields over an

integer *m* (`mod m`). Also, there are the container types, which can be vectors, matrices or structures. On the other hand, the Cryptol's type system is composed by the Bit type, which then can be used to create sophisticated data types, including vectors or matrices, by the use of sequences.

In the Table 5.1, the relation between the representation of CAO's types in Cryptol is presented. The ϵ symbol, represents the size in bits of a CAO data type.

Type in CAO	**Type in Cryptol**
void	_
int	ϵ
bool	True or False : [1]
signed bits[n]	[ϵ + 1]
unsigned bits[n]	[ϵ]
mod m	
vector[n] of a	[n][ϵ]
matrix[n,m] of a	[n] [m] [ϵ]

Table 5.1: CAO's types relation with Cryptol's types

At first glance, the *int*, *bool*, *signed bits* and *unsigned bits* can be represented in Cryptol, because the values of these types can be represented using bits. The void type is usually used as a return type for functions that produce side effects. As Cryptol is a functional language that does not support them, such type will not be considered. The finite field type, will not be considered for compilation, at this point.

The CAO language provides vectors and matrices that can store a set of values of the same type. These container types may be represented in Cryptol as sequences. Structures are also containers that may contain variables of distinct types. One approach would be to represent them using tuples or records in Cryptol. For this purpose, tuples will be used in Cryptol to represent CAO's structures.

Function signatures

A CAO function is defined by the function signature, which is composed by the function name, the arguments (variable and variable type), its return type and a list of statements.

Listing 5.1: A sample function in CAO

```
1 def func() : unsigned bits[163] {
2   (...) /* list of statements */
3 }
```

On the other hand, Cryptol does not require that the function signatures are explicitly defined, however, it is a good practice to define them. The previous function would be represented in Cryptol as the following:

Listing 5.2: A sample function in Cryptol

```
1 func : [163];
2 func = (...) /* list of statements */
```

Operations

The two languages provide almost the same operators, which can be divided into the following groups: arithmetic, boolean, equality, comparison, shifting and rotation.

The operations provided by both languages are similar, allowing the developer to evaluate expressions such as $e_1\ op\ e_2$, where op represents a binary operator. Most of these binary operators, have a direct correspondent in the other language:

Arithmetic operators The supported set of arithmetic operators of the two languages its almost the same. Operators such as sum (+), subtraction (-), multiplication ($*$) or division ($/$) operators are the same in the two languages. The modulo (%) and the $**$ operators are also supported by both languages. However, the Cryptol language supports the `lg2` operator that CAO does not.

Boolean operators The boolean operators *and* (`&&`), *or*, *exclusive-or* are supported by both languages, although they are represented by distinct symbols. The Cryptol language also supports the (~) (complement) and the `negate` operator, that CAO does not.

Equality and comparison operators Equality testing is supported by both languages, and are represented by the same operators: == (equal test), != (not equal test), < (less), > (more), >= (less or equal), <= (more or equal).

Shifting and Rotates Bit shifting and rotate functions exist in both languages. The shifting operators << (left shifting) and >> (right shifting) are represented in the same way in CAO and Cryptol. The CAO's rotate operators <| (left rotate) |> (right rotate) are represented by the Cryptol's <<< and >>> operators, respectively.

Container element selection The selection of an element of a container through an index (an index if it is a vector or a field name if it is a structure) is possible in Cryptol. However, multiple selection using ranges in CAO, which may be retargeted using the @@ function in Cryptol, will not be considered for code compilation.

Conditional Expressions and Other considerations

As detailed in the Chapter 2, both languages support conditionals: the CAO language supports if-then and if-then-else compound statement; on the other hand, the Cryptol language, supports if-then-else expressions which do not have produce any side effect in the program. However, as CAO is an imperative language, the branches of the if-then-else statement may be any other statement that can change the state and the variables of the program.

The iteration constructs of CAO, such as `while`, also have this problem. For the sake of simplicity they will not be considered in the compiler CAO2Cryptol. The `seq` and `seq by` statements, can be expanded: the number of executions of each cycle must be known at compile, and so, the code may be replicated. For example,

Listing 5.3: Example of `seq` in CAO

```
1 seq i := 0 to 4 {
2   j := j+1;
3 }
```

can be expanded to:

Listing 5.4: Example of an expanded `seq` in CAO

```
1 j := j +1;
2 j := j +1;
3 j := j +1;
4 j := j +1;
```

The type conversions or casts, possible in CAO for some data types, will not be considered for the compiler. Another aspect, is that a CAO program may have multiple return statements in a function. Although a tuple may be returned (with multiple values), a function must return only once.

5.1.2 Compiler Architecture

In order to re-target a source file written in the CAO language to Cryptol, in a way that such program can be processed and generate VHDL code, it is necessary that the generated code take into account the limitations of the Cryptol back end for hardware languages.

In fact, some features of Cryptol does not compile to hardware. It is necessary to be aware of such features, and assure that the generated Cryptol code does not use them. As CAO does not support polymorphic types (only the CALF compiler extension supports them), it automatically excludes the use of polymorphism in the generated code: every function or variable in CAO is monomorphic and has its type explicitly declared. However, CAO's variables may change during the execution of the program.

Listing 5.5: Example of a CAO function

```
1  def f(a : int) : int
2  {
3     def x : int;
4     if (a > 0) {
5        x :=4
6     } else {
7        x := 2∗a;
8     }
9     return x;
10 }
```

In this Listing 5.5, the variable `x` is modified in line 5 and 7. Actually, Cryptol does not support this: each variable must be defined only once in a function. Also, the if-then-else statements must be simplified in order to be translated into Cryptol's if-then-else expressions.

To solve these problems, the CAO input program will be transformed into the SSA form, which states that each variable is defined once and only once in the program [AP02]. To calculate the SSA form of a CAO program, it is necessary to calculate the CFG of a CAO program. The is a graph representing the instructions of the CAO program. After this calculation, the dominance frontier algorithm is applied to the CFG, allowing the calculation of the SSA form. Afterwards, the procesed CAO program is translated to Cryptol, using a pretty printer module. These algorithms will be approached in detail in the following sections.

5.2 Compiler Implementation

In this section, the required transformations for the CAO2Compiler will be presented, namely the CFG, the set of algorithms for the dominance frontier calculation and the SSA form.

5.2.1 Control Flow Graph

To perform analysis and optimisations on a program, in this case in a CAO program, it is useful to generate a control-flow graph (CFG) [All70] [AP02]. The CFG is a directed graph, in which each statement of the program is a node in the flow graph and if statement `x` is followed by statement `y`, then there is an edge connecting `x` and `y`. However, nodes with only one predecessor and one successor can be merged into basic blocks [AP02]. The result is a control-flow graph with fewer nodes, making the execution of algorithms over the graph to execute faster. Consider the following pseudo-code example:

Listing 5.6: Example to illustrate the CFG calculation

```
1 y<−0
2 if (X>1)
3    y <− x
4    x <− 2
5 else
6    y <− 2
7 return y
```

The CFG calculated based on this program, can be represented as the following graph:

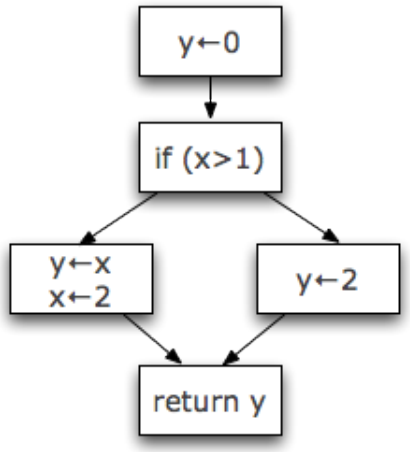

Figure 5.1: Control-flow graph

In the Figure 5.1, the CFG has four nodes. Note that the if-then-else conditional creates two branches, one for the then clause and one for the else clause, and finally the fourth node, which merges both branches.

After calculate the CFG, it is now possible to execute algorithms over graphs, most of them used in program optimization, such as the dominance frontier algorithm.

5.2.2 Dominance Frontier

Before present the dominance frontier algorithm, it is important to introduce basic concepts related to it, such as the concept of predecessor, successor, dominator and dominator tree.

Dominators

A node *n* in the CFG is said to dominate *b* if *n* passes through node *b* for every path, from the entry node of the CFG. The set of dominators for a node *b* is known as $DOM(b)$ and contains everynode *n* that dominates *b* [CHK01]. Also, a node *n* strictly dominates *b* if *n* dominates *b* and $n \neq b$.

As described in [CHK01], the algorithm to calculate the dominators of a graph such a CFG, is the following:

for all nodes n
    DOM[n] ← { 1 . . N }
Changed ← true
while (Changed)
    Changed ← false
    for all nodes, n, in reverse postorder
        new_set ← $(\cap_{p \in preds(n)} DOM[p]) \cup \{n\}$
        if (new_set ≠ DOM[n])
          DOM[n] ← new_set
          Changed ← true

Figure 5.2: Iterative Algorithm to calculate dominators for each CFG node [CHK01]

Dominator Tree

In a dominator tree, the children of a node *x* are the set of all nodes that *x* immediately dominates.

The algorithm to calculate the dominator tree, based on a graph such a CFG and the dominators, is represented in the Figure **??**, as described in [CHK01].

This algorithm takes as input a graph (for instance, a CFG), and returns the dominator tree of that graph. The dominator tree is useful to calculate the dominance frontier, which can be further used to calculate the SSA form of a program.

Dominance Frontier Algorithm

The dominance frontier algorithm, presented in the Figure 5.4, is calculated for each node of the CFG and it is useful to calculate the SSA form of a program.

```
for all nodes, b
    doms[b] ← Undefined
doms[start_node] ← start_node
Changed ← true
while (Changed)
    Changed ← false
    for all nodes, n, in reverse postorder
        new_idom ← first (processed) predecessor of b
        for all other predecessors, p, of b
            if doms[p] ≠ Undefined
              new_idom ← intersect(p, new_idom)
        if doms[b] ≠ new_idom
            doms[b] ← new_idom
            Changed ← true

function intersect(b1, b2) returns node
    finger1 ← b1
    finger2 ← b2
    while (finger1 ≠ finger2)
          while (finger1 < finger2)
                finger1 = doms[finger1]
          while (finger2 < finger1)
                finger2 = doms[finger2]
    return finger1
```

Figure 5.3: Iterative Algorithm to calculate dominators for each CFG node [CHK01]

In [CFR+91], dominance frontier of a node *x* is defined as the set of all nodes *y* such that *x* dominates a predecessor *y* but does not strictly dominate *y*.

```
for all nodes, b
    if the number of predecessors of b ≥ 2
      for all predecessors, p, of b
          runner ← p
          while runner ≠ doms[b]
                add b to runner's dominance frontier set
                runner = doms[runner]
```

Figure 5.4: The dominance frontier algorithm [CHK01]

The calculation of the dominance frontiers for each node in the CFG plays an important role in the single-static assignment form. After this calculations, the SSA form of a CAO program can be calculated.

5.2.3 Single-static Assignment Form

The SSA form states that each variable has only one definition in the program [AP02]. It allows the execution of dataflow and optimization algorithms, that are simpler when each variable has only one definition. Usually, a program is translated to the SSA form, processed and optimized and then is translated back out from from the SSA form [CFR+91].

For example, the pseudo-code presented in the Figure 5.1, would have the following representation in the SSA form:

```
1 y_0 <−0
2 if (X>1)
3   y_1 <− X
4   x_1 <− 2
5 else
6   y_2 <− 2
7 return phi_y(y_1,y_2)
```

The SSA form is divided into two major steps:

1. insertion of ϕ-functions for each variable at each join point
2. renaming the definitions and uses, using subscripts

Inserting ϕ-functions

The first step of the SSA form, starts with a set V of variables, a graph G (the CFG) with program nodes (each node is a basic block of statements) and for each node *n*, is calculated a set A$_{o}rig$[n], with the name of variables defined in that node. The pseudo-code of the algorithm to insert ϕ-functions is described in the Figure 5.5.

The algorithm computes Aϕ[*a*], the set of nodes that must have ϕ-functions for the variable *a*. In practice, ϕ-functions are inserted after join points, which ocurr essentially when there are conditional branches in the control-flow graph, such as `if` or `while` statements.

```
for each node n
    for each variable a in A_orig[n]
        defsites[a] ← defsites[a] ∪ {n}
for each variable a
    W ← defsites[a]
    while W not empty
        remove some node n from W
        for each y in DF[n]
            if y ∉ A_φ[a]
              insert the statement a ← φ(a, a, ..., a) at the top
                    of block y where the φ-function has as many
                    arguments as y has predecessors
              A_φ[a] ← A_φ[a] ∪ {y}
              if a ∉ A_orig[y]
                    W ← W ∪ {y}
```

Figure 5.5: Inserting ϕ-functions step of the SSA form

Renaming variables

The second step of the SSA form is to rename each occurrence of a variable, such as *a*, to *a_1* or *a_2*. The main purpose of this step is that the program uses the most recent version of `a`. For this purpose, it is used the dominance frontier, calculated previously.

This algorithm is divided into two steps: the first initializes two auxiliary stacks, for each variable: Count and Stack; the second step is the algorithm that transverses each statement of the program to rename the variables.

```
for each variable a
    Count[a] ← 0
    Stack[a] ← empty
    push 0 onto Stack[a]
```

This algorithm, transforms a program given as input into the correspondent SSA form. After this transformations, the program may now be processed or optimized. The most common optimizations are code motion, elimination of partial redundancies or constant propagation [CFR+91].

```
Rename(X) =
for each statement S in block n
    if S is not a φ-function
      i ← top(Stack[i])
      replace the use of x with x_i in S
    for each definition of some variable x in S
      Count[a] ← Count[a] + 1
      i ← Count[a]
      push i onto Stack[a]
      replace definition of a with definition of a_i in S
for each successor Y of block n
    Suppose n is the j^th predecessor of Y
    i ← top(Stack[a])
    replace the j^th operand with a_i
for each child X of n
    Rename(X)
for each definition of some variable a in the original S
    pop Stack[a]
```

Figure 5.6: Renaming variables (SSA form, step 2

Container types

CAO support container types such as vectors, matrices or structures. One of the most used operations is the access to a specific element of a container, using an index (in a vector) or the field identifier (in a structure).

Most part of the mentions of vectors and structures in a CAO program is to access specific indexes or fields. Considering a vector `v` in CAO, we can access the index `i` through `v[i]`. Actually, this can be used as a scalar variable, like an `int` or `bool`. However, as we need to transform the CAO program into the SSA form, these containers must be seen as an unique variable [CFR+91].

For example, consider the following code snippet, which uses a vector with eight integers and an auxiliary variable, to execute a swap between two vector elements:

In the line 4, there is an assignment that reads the value in the position 1 of `v` and assigns it to `v`. In the following line, there is a similar assignment that reads a value of `v` and assigns it to the element of `v` in the index 2. In the line 6, there is another assignment, that stores the value of `x` in `v[2]`.

Listing 5.7: Swap operation in CAO using a vector

```
1 def v : vector[8] of int := {1,2,3,4,5,6,7,8};
2 def x : int;
3
4 x := v[1];
5 v[1] := v[2];
6 v[2] := x;
```

With the approach detailed in [CFR+91], the array may be treated as single variable. When a occurs a modifitation in an element of the container, the whole memory (vector or structure), should be replicated. The above example, would be replaced with:

Listing 5.8: Swap operation in CAO using a vector after the transformation

```
1 def v : vector[8] of int := {1,2,3,4,5,6,7,8};
2 def x : int;
3
4 x := load(v,1);
5 v := store(v,1(load(v,2));
6 v := store(v,2,x);
```

The structures in CAO are also containers, However, instead of indexes there are fields that can be accessed, similar to what happens in vectors or matrices. Considering the same swap operation, but this time using a structure:

Listing 5.9: Swap operation in CAO using a structure

```
1 typedef GFPoint := struct [def cx : int; def cy : int;];
2 def s : GFPoint;
3 def x : int;
4
5 x := s.cx;
6 s.cx := s.cy;
7 s.cy := x;
```

The same *store* and *load* functions may be applied, resulting in the following output:

Listing 5.10: Swap operation in CAO using a structure after the transformation

```
1 typedef GFPoint := struct [def cx : int; def cy : int;];
2 def s : GFPoint;
3 def x : int;
4
5 x := load(s,cx);
6 s = store(s,cx,load(s,cy));
7 s = store(s,cy,x);
```

After these transformations, applied to container types (vectors, matrices and structures), the SSA form may now be applied, because such variables are now treated as scalar variables.

5.2.4 Other transformations

The transformations presented here, aim to assure that a variable is defined only once in a program. Also, the ϕ-functions are inserted essentially when conditional branches such as if-then-else occur. However, in order to eliminate the if-then-else statements, a few transformations are required: initially, the conditions of each branch are inserted directly into the ϕ-functions; later, the if-then and if-then-else statements are removed, assuring that their statements remain in the program. With this transformation, a function of a program may be now seen as an unique basic block in the CFG.

Insert phi-conditions

This transformation, aims to insert in each ϕ-function, the condition that allows the program to decides if it takes the first argument or the second one from the ϕ-function.

Considering the code snippet in the Listing 5.11, the value of y_3 would depend on the value of `a`: if `a` is bigger than 1 then y_3 takes the value of y_1, otherwise, it takes the value of y_2. This transformation aims to introduce the condition into the ϕ-function, resulting in the code presented in the Listing 5.12.

Listing 5.11: Example of CAO code in the SSA form

```
1 if (a > 1) {
2    y_1 := 1;
3 } else {
4    y_2 := 10;
5 }
6 y_3 := phi_y(y_1,y_2)
```

Listing 5.12: Example of CAO with the condition implicit in the ϕ-function

```
1 if (a > 1) {
2    y_1 := 1;
3 } else {
4    y_2 := 10;
5 }
6 y_3 := phi_y(y_1,y_2, a> 1)
```

Remove if-then-else statements

This transformation, aims to take on the output of the previous one, and remove the if-then-else statements, always assuring that its statements remain in the program.

Taking the previous example, this transformation results as output:

Listing 5.13: CAO code in an unique basic block

```
1    y_1 := 1;
2    y_2 := 10;
3    y_3 := phi_y(y_1,y_2, a> 1)
```

These programs are equivalent, essentially because the `phi_y` function stores the condition to decide which value will take y_3. After these transformations, a CAO function may be seen as an unique basic block of straight-line code.

5.3 CAO2Cryptol Pretty Printer

The pretty printer module of the CAO2Cryptol compiler, aims to take on a CAO program modified according to the transformations described previously, and pro-

duce Cryptol code that can be interpreted by the Cryptol tools.

The CAO code that is given as input to the pretty printer module is composed by a basic block, containing only declarations, assignments and return statements: the if-then-else statements were removed in the SSA form transformations. In this section, we will explain the main phases of the translation of a CAO program to Cryptol.

In the AST of the CAO language, there are three types of definitions, which can be variables, types or functions: the variables are directly translated to Cryptol as a function; the types are translated to Cryptol as a new type definition; the functions are translated according to the Listings 5.14 and 5.17

The two kinds of variable declaration that initializes a variable in a single declaration (`VarDeclInit` and `ContainerDeclInit`) require two lines in Cryptol: one to the function signature and another to the assignment of the initialization.

Data types The data types in CAO are translated to Cryptol according to the Table 5.1:

- *int* is translated into [1024]
- *bool* is translated into [bool]
- *unsigned bits[n]* is translated into [n]
- *signed bits[n]* is translated into [n]
- *vector[n] of a* is translated into [n][(size of a)]
- *matrix[n,m] of a* is translated into [n][m][(size of a)]

The void type is not possible to be implemented in a functional language, and as stated previously, the finite fields were not considered for compilation.

Variable Declaration The variable declarations in CAO may be of four types:

- `def x : bool;` is translated into `x : [1];`
- `def x,y : bool;` is translated into `x,y : [1];`
- `def x : int := 10;` is translated into `x : [1024]; x = 10;`

Function Definition The translation of a CAO function, which is composed by the function name, its arguments and respective types, the return type and the list of statements is made accordingly to the Listing 5.14 and 5.17:

A function definition in CAO is composed by the function name, its arguments and its types, the return type and the list of statements. The following function:

Listing 5.14: A sample function in CAO

```
1 def func() : unsigned bits[163] {
2   (...) /* list of statements */
3 }
```

is translated as:

Listing 5.15: A sample function in Cryptol

```
1 func : [163];
2 func = (...) /* list of statements */
```

However, in the body of the function (the list of statements) of the CAO program to be translated, can appear assignments and declarations. Although the if-then-else control structures have been removed, it still can appear function calls of ϕ-functions.

A declaration and an assignment are translated almost directly: the declaration declares a new variable with the correspondent type in the Cryptol language; the assignment binds the value to the variable on the left side of the expression.

The ϕ-functions, usually appear as following:

Listing 5.16: ϕ-function in CAO

```
1 y_3 := phi_y(y_1,y_2,x > 1);
```

The translation to Cryptol, is direct, requiring only to use the properly syntax:

Listing 5.17: ϕ-function translation for Cryptol

```
1 y_3 = if (x> 1) then y_1 else y_2;
```

Another relevant part of the translation are the container types: vectors, matrices and structures. As stated previously, vectors and matrices are translated to sequences. On the other hand structures in CAO are translated to tuples in Cryptol, which allow the representation of different data types in the same variable.

For example, considering the following code snippet:

Listing 5.18: Translating vectors to sequences (CAO version)

```
1 def v : vector[8] of bool;
2
3 /* accessing a position in a function */
4 def a := v[5];
5
6 /* storing a new value in a position, in a function */
7 v[0] := 10;
```

would be translated to the following:

Listing 5.19: Translating vectors to sequences (Cryptol version)

```
1 v : [8][1];
2
3 /* accessing a position in a function */
4 a = v @ 5;
5
6 /* storing a new value in a position, in a function */
7 a = take(0,v) # [10] # drop(0+1,v);
```

The structures in CAO are translated into a tuple, where each element has the same type of the original type in the CAO program. For example, a structure with two booleans, is translated to `type D = ([1],[1]);` in Cryptol. Also, a *load* and a *store* function is generated, to access each element of the structure.

5.4 Summary

In this chapter, we proposed a tool to compile CAO source code to Cryptol, assuring that the output code does not use unsupported feature by the VHDL back end of the Cryptol toolkit. In the first section, it was described the requirements

for the compiler, focusing on the limitations of the Cryptol toolkit and the feature of the two languages, in order to identify its similarities.

The Cryptol code must be straight-line and a variable must be defined only once, unlike CAO. This made the necessity to transform the CAO program to the SSA form before translate it to Cryptol. The if-then or if-then-else statements, are also distinct in the two languages: in CAO they are statements which can modify the whole state of the program and in Cryptol they are expressions which must return the same type. By using the dominance frontier algorithms and the SSA form, this problem was solved, and made the CAO code translatable.

After apply a set of transformations to the original CAO code given as input to the compiler, it is translated to Cryptol by the use of a pretty printing module. This module generates as output Cryptol compilable code, which can then be translated to VHDL.

In the next chapter, we will address a case study based on elliptic curve cryptography, comparing an implementation of an algorithm in both languages, and testing the developed tool.

Chapter 6

Case study: elliptic curve cryptography

Through this document, we addressed the features that Cryptol, CAO and its toolkits. So far, the AES algorithm was presented, with particular focus on its encryption scheme. The AES algorithm was useful to compare two implementations of its encryption scheme in the two DSLs, which are distinct, mostly because of their paradigms. The compilation tools, specially the ones that allow the re-targeting of Cryptol code to VHDL and FPGA board platforms were studied in Chapter 4. In the last chapter, a tool that aims to take on CAO source file as input, and generates as input Cryptol source code as presented. Also, it is intended that the generated code can be translatable to VHDL, using the Cryptol toolkit.

The elliptic curves are algebric constructions defined over prime or binary finite fields and were proposed by Neal Koblitz in [Kob87], to implement public-key algorithms. In [Mil86], the Diffie-Hellman key-exchange algorithm was solved using elliptic curves and it is proved that such structures may be used to implement cryptographic algorithms. One of the biggest advantages of these systems is that they may use smaller key sizes to achieve the same security level when compared to classical finite fields [Sch93]. With smaller key sizes, elliptic curves may be useful for hardware implementations, where the computer resources are rare.

In this chapter, we present a case study, which focuses on elliptic curve cryptography. After present a brief summary on elliptic curves and its application to cryptography, we present two implementations of an operation of elliptic curves in CAO and Cryptol. As the two languages differ in their paradigms and features,

we will focus on the pros and cons of re-targeting an algorithm to an imperative or a functional language. Finally, using the developed tool presented in the Chapter 5, CAO2Cryptol, we will test the compiler, using an implementation written CAO as input and analysing the Cryptol code that result as output.

6.1 Elliptic Curve Cryptography

The elliptic curves may be used to implement a large set of public-key primitives, such as digital signatures. One of these primitives is the Elliptic Curve Digital Signature Standard (ECDSA), for instance.

The elliptic curve algorithms use finite fields as its coefficients for the equation of the curve. Also, each point of the curve P is represented by the coordinates (x, y and z), which are also members of the same finite field. In cryptography, two types of finite fields are used: the prime finite fields GF(p) and the binary finite fields GF(2^m). In the former, p is a prime number, and the set Z_p is a prime finite field. In the latter, the binary finite field GF(2^m) represents the 2^m possible bit strings of length m.

In this work, we will use elliptic curves with binary finite fields on its coefficients and points. The addition of two values in a binary finite field is made by calculating the bitwise addition modulo 2. The multiplication in a binary finite field is the multiplication of two values modulo an irreducible polynomial, used to define the finite field.

The generic elliptic curve equation, for the ones that use binary finite fields is the following:

$$E: y^2 + xy = x^3 + x^2 + b$$

where b is an element of GF(2^m) with $b \neq 0$.

Through this work, we will use the binary finite field GF(2^{163}) which is part of the FIPS 186-2 standard [HLHM00]. The recommended values by NIST for the B-163 parameters of the elliptic curve are the following:

- n = 5846006549323611672814742442876390689256843201587

- b = 20a601907b8c953ca1481eb10512f78744a3205fd
- G_x = 3f0eba16286a2d57ea0991168 d4994637e8343e36
- G_y = 0d51fbc6c71a0094fa2cdd545b11c5c0c797324f1

Also, the irreducible polynomial used in this by the binary finite field is $p(t) = t^{163} + t^7 + t^6 + t^3 + 1$.

Generically, the elliptic curves allow one to implement cryptographic public-key primitives. The similarities between elliptic curves and finite fields that use the discrete logarithm, are the following:

	DL	**EC**
Setting	GF(q)	curve E over GF(q)
Basic operation	multiplication in GF(q)	addition of points
Main operation	exponentiation	scalar multiplication
Base element	generator g	base point G
Base element order	prime r	prime r
Private key	s (integer modulo r)	s (integer modulo r)
Public key	w (element of GF(q))	W (point on E)

Table 6.1: Comparison between Discrete Logarithm and Elliptic Curves [IEE99]

The main operation over elliptic curves, is the scalar multiplication, as we can see in the Table 6.1. However, such operation uses other minor operations, such as the Full Addition, Subtraction, Double and Addition. Also, there is a representation of the point in the infinity, which is represented by the point with coordinates (1,1,0). Through this section, the formal specification of the Double operation will be presented, in order to provide the required background to the implementations in CAO and Cryptol, ahead in this chapter.

6.1.1 Elliptic Double operation

The Double algorithm, described in [IEE99], takes on a point $P_1 = (X_1, Y_1, Z_1)$ of the elliptic curve in which its coordinates x, y and z are elements of 2^{163}; Also, this algorithm uses a constant `c` which is calculated from the original parameter `b`, using the following equation: $b^{2^{163-2}}$. The formal specification of this operation has an imperative approach and is presented in the Figure 6.1.

1. $T_1 \leftarrow X_1$
2. $T_2 \leftarrow Y_1$
3. $T_3 \leftarrow Z_1$
4. $T_4 \leftarrow$ c
5. If T_1 = 0 or T_3 = 0 then output (1,1,0) and stop.
6. $T_2 \leftarrow T_2 \times T_3$
7. $T_3 \leftarrow {T_3}^2$
8. $T_4 \leftarrow T_3 \times T_4$
9. $T_3 \leftarrow T_1 \times T_3$
10. $T_2 \leftarrow T_2$ + T_3
11. $T_4 \leftarrow T_1$ + T_4
12. $T_4 \leftarrow {T_4}^2$
13. $T_4 \leftarrow {T_4}^2$
14. $T_1 \leftarrow {T_1}^2$
15. $T_2 \leftarrow T_1$ + T_2
16. $T_2 \leftarrow T_2$ + T_4
17. $T_1 \leftarrow {T_1}^2$
18. $T_1 \leftarrow T_1 \times$ T_3
19. $T_2 \leftarrow T_1$ + T_2
20. $T_1 \leftarrow T_4$
21. $X_2 \leftarrow T_1$
22. $Y_2 \leftarrow T_2$
23. $Z_2 \leftarrow T_3$

Figure 6.1: EC Double algorithm using projective coordinates

The output of this operation, is the point $P_2 = (X_2, Y_2, Z_2)$, which is the double of the point P_1 given as input.

6.2 Elliptic Curve Implementations

In this section, we will present the implementation of elliptic curves in CAO and Cryptol. The main focus will be on the data types to be used, as well as the Double operation over an elliptic curve point.

6.2.1 Implementation in CAO

The required variables to execute an elliptic curve algorithm are the points of the curve and its parameters. Each point P is composed by its coordinates. The B-163 curve uses 163-bit words to represent its points' coordinates and its

parameters, which can be represented by the type `unsigned bits[163]`. In the CAO language, there are a few alternatives to represent each curve point:

- a structure with three fields (*x*, *y* and *z*)
- a vector with three positions (position 0,1,2 for *x*, *y* and *z*, respectively)

Using the first alternative, a structure defining a new type `GFPoint` with three fields for each point of an elliptic curve, may be defined as follows:

Listing 6.1: Elliptic curve point in CAO

```
1 typedef GFPoint := struct [def x : unsigned bits[163]; def y : unsigned bits[163]; def z :
      unsigned bits[163];];
```

The same `GFPoint` would be the second alternative, the vector with three positions, could be defined as:

Listing 6.2: Elliptic curve point in CAO

```
1 typedef GFPoint := vector[3] of unsigned bits[163];
```

However, using bit vectors to define each element of GF(2^{163}), the addition and multiplication operations must be explicitly defined for that finite field:

Listing 6.3: Elliptic curve addition and multiplication

```
1 def mult (def a : unsigned bits[163], def b : unsigned bits[163], def irred : unsigned bits
      [163]) : unsigned bits[163] {
2   def c : unsigned bits[163];
3   c := (a * b) \% irred;
4   return c;
5 }
6
7 def sum (def a : unsigned bits [163], def b : unsigned bits[163]) : unsigned bits[163] {
8   return a ^ b;
9 }
```

To note that the irreducible polynomial defined for the elliptic curve B-163 is an argument of the multiplication function, and is used to assure that the operation is made correctly.

Another alternative to define the bit strings for each coordinate of the elliptic curve point is through the use of the mod type that exist in CAO. Such structure allows one to define a finite field based on a polynomial, like the one used in the B-163 curve. With that type, the addition and multiplication may be done by using the $+$ and $*$ operators, as these operations are natively implemented in the CAO language.

Elliptic Double operation

The `Double` function takes as input a point P of the elliptic curve, and after execution it returns a new point, which corresponds to $2P$. One implementation of the `Double` operation described in the Figure 6.1, could be the following code:

Listing 6.4: EC2Double in CAO

```
1  def c : unsigned bits[163] := (...) // c := b**(2**(163-2))
2
3  def EC2Double (P : GFPoint) : GFPoint
4  {
5    def nP : GFPoint;
6    def T1,T2,T3,T4 : unsigned bits[163];
7    T1 := P.x;
8    T2 := P.y;
9    T3 := P.z;
10   T4 := c;
11   if ( T1 == 0b0 || T3 == 0b0 ) {
12     return EC2infinity();
13   }
14   T2 := mult(T2,T3);
15   T3 := mult(T3,T3);
16   T4 := mult(T3,T4);
17   T3 := mult(T1,T3);
18   T2 := sum(T2,T3);
19   T4 := sum(T1,T4);
20   T4 := mult(T4,T4);
21   T4 := mult(T4,T4);
22   T1 := mult(T1,T1);
23   T2 := sum(T1,T2);
```

```
24    T2 := mult(T2,T4);
25    T1 := mult(T1,T1);
26    T1 := mult(T1,T3);
27    T2 := sum(T1,T2);
28    T1 := T4;
29    nP.x := T1;
30    nP.y := T2;
31    nP.z := T4;
32    return nP;
33 }
```

The implementation in CAO of this algorithm based on its formal specification is almost direct. This implementation has two return points: the first at the line 10 (that only returns if the coordinates x and z are zero; the second is done at the end of the function. Furthermore, the auxiliary functions `sum` and `mult` are used, in order to execute the addition and multiplication in the finite field GF(2^{163}).

6.2.2 Implementation in Cryptol

In order to represent the required variables for the elliptic curve operations (the coordinates of a point and the parameters of the elliptic curve), there are different approaches that may be used in the Cryptol implementation.

The first alternative for the implementation of the points $P = (x, y, z)$ is through the use of a tuple with three elements of the type `([163],[163],[163])`: three elements represented by 163 bits each. The second one, is through the use of a sequence with three elements, where the positions 0, 1 and 2 represent the x, y and z coordinates, respectively. The last alternative, is to use a record, which can be defined as follows:

Listing 6.5: Elliptic curve point in CAO using records

```
1 type GFPoint = {x : [163]; y : [163]; z : [163]};
```

However, independently of the chosen alternative, it is necessary to define the addition and multiplication operations, similar to the CAO's implementation:

Listing 6.6: Elliptic curve addition and multiplication in Cryptol

```
irred = <| x^163 + x^7 + x^6 + x^3 + 1 |>;
mult (a,b) = pmod (pmult(a,b),irred)

sum : ([163],[163]) -> [163]
sum (a,b) = s
  where {
    x = a : [163];
    y = b : [163];
    s = x ^ y;
  };
```

The `irred` variable represents the polynomial used in the B-163 curve and is used in the multiplication function.

Elliptic Double Operation

The implementation of the algorithm introduced in the Figure 6.1 in Cryptol, is the following:

Listing 6.7: EC2Double in Cryptol

```
EC2Double : GFPoint -> GFPoint;
EC2Double p = newPoint
  where {
    t1 = getX p;
    t2 = getY p;
    t3 = getZ p;
    t4 = c;
    t2' = mult t2 t3;
    t3' = mult t3 t3;
    t4' = mult t3 t4;
    t3'' = mult t1 t3';
    t2'' = sum t2' t3'';
    t4'' = sum t1 t4';
    t4''' = mult t4'' t4'';
    t4'''' = mult t4''' t4''';
    t1' = mult t1 t1;
    t2''' = sum t1' t2'';
```

```
18      t2'''' = mult t2''' t4'''';
19      t1''  = mult t1' t1';
20      t1''' = mult t1'' t3'';
21      t2''''' = sum t1''' t2'''';
22      t1'''' = t4'''';
23      newPoint = (t1'''',t2''''',t3'');
24    };
```

Although Cryptol is a functional language, this implementation looks like an imperative one. It uses the *where* clause, that allows one to bind values to functions/variables. However, as each variable must be defined only once in the execution of a program, "'" are concatenated to the variables.

6.3 Compilation using the CAO2Cryptol tool

The compilation tool presented in the Chapter 5, aims to take on CAO source code given as input and translate it to Cryptol. The Cryptol generated as output should not use polymorphism and other unsupported constructs by the Cryptol's back ends, so that it can be translatable to VHDL.

Considering the CAO's implementation of the `EC2Double` function, it has two return statements in the function. In order to translate it to Cryptol using the developed tool, it must have only one return statement. Also, the return statement should be the last statement of the function, returning the last state of the returning variable.

Therefore, the CAO code presented in the previous section must be transformed so that it uses only one return statement: a new variable named `nP` with the state of the return statement is created. Then, it is returned in the last statement of the function.

For example, using structures, the code necessary to access (load) and modify (store) the values of each coordinate of the point would be the following:

Listing 6.8: Structure load and store operations in the EC2Double function

```
1
2  /* (...) */
```

```
3  /* load */
4     T1 := P.x;
5     T2 := P.y;
6     T3 := P.z;
7
8  /* EC2Double algorithm */
9
10 /* store */
11    nP.x := T1;
12    nP.y := T2;
13    nP.z := T4;
14
15 /* return statements */
```

Furthermore, the CAO2Cryptol compilation tool only supports local variables, i.e. it does not support a variable declared out of the scope of the function. The `EC2Double` function uses the constant `c`, which can be passed as argument or be defined inside the function scope. In this implementation, we defined the `c` variable explicitly in the function.

After executing the `EC2Double` compiler, giving the code above as input, it succeeds. However, when trying to compile the output to VHDL, the follow error occurs, due to the use of tuples.

Listing 6.9: Compilation error using Cryptol-VHDL

```
1  "ECurves.cry", line 14, col 1: unable to compile use of '%%LAMBDA%%_store_s_x.
       _163' as a function value -- its definition isn't closed.
```

However, a new approach using the proposed solution which uses a vector with three elements of the type `unsigned bits[163` was more successful. After transforming the `EC2Double` according that structure, the following code was given as input to the CAO2Cryptol compiler.

Listing 6.10: EC2Double input for the CAO2Cryptol compilation tool

```
1  typedef GFPoint := vector[3] of unsigned bits[163];
2  def EC2Double (P : GFPoint) : GFPoint
3  {
4     def nP : GFPoint := EC2zero();
5     def T1,T2,T3,T4 : unsigned bits[163];
6     def c : unsigned bits[163] := (...) // c := b**(2**(163-2))
7     T1 := P[0];
8     T2 := P[1];
9     T3 := P[2];
10    T4 := c;
11    if ( (T1 == 0b0) || (T3 == 0b0) ) {
12       nP := EC2infinity();
13    } else {
14    T2 := mult(T2,T3);
15    T3 := mult(T3,T3);
16    T4 := mult(T3,T4);
17    T3 := mult(T1,T3);
18    T2 := sum(T2,T3);
19    T4 := sum(T1,T4);
20       T4 := mult(T4,T4);
21    T4 := mult(T4,T4);
22    T1 := mult(T1,T1);
23    T2 := sum (T1,T2);
24    T2 := mult(T2,T4);
25    T1 := mult(T1,T1);
26    T1 := mult(T1,T3);
27    T2 := sum(T1,T2);
28    T1 := T4;
29    nP[0] := T1;
30    nP[1] := T2;
31    nP[2] := T4;
32    }
33    return nP;
34 }
```

The total output of the translation can be checked in the Appendix B, however, a

code sample of the output is presented in the Listing 6.11.

Listing 6.11: EC2Double output sample using the CAO2Cryptol tool

```
1  EC2Double : (GFPoint) -> GFPoint;
2  EC2Double (P) = nP_5
3          where {
4    /* (...) */
5                  T1_1 = mult(T1_0, T1_0);
6                  T2_3 = sum(T1_1, T2_2);
7                  T2_4 = mult(T2_3, T4_4);
8                  T1_2 = mult(T1_1, T1_1);
9                  T1_3 = mult(T1_2, T3_2);
10                 T2_5 = sum(T1_3, T2_4);
11                 T1_4 = T4_4;
12                 nP_2 = take (0,nP_1) # [T1_4] # drop (0+1,nP_1);
13                 nP_3 = take (1,nP_2) # [T2_5] # drop (1+1,nP_2);
14                 nP_4 = take (2,nP_3) # [T4_4] # drop (2+1,nP_3);
15                 nP_5 = if ((T1_0 == 0b0) | (T3_0 == 0b0)) then (nP_1) else (nP_4);
16       /* (...) */
17         };
```

Considering the output, it is important to note a few key points:

- The load of a position of the vector is translated to operator and the store operation is translated using the take and drop functions of the Cryptol language.
- For each store operation on the vector, it is necessary to replicate the memory used by the memory.
- For each variable and each definition of a new value, it is created a new variable with the respective index as subscript.
- As the variables were typed in CAO, they still have its type represented in the Cryptol implementation.

This code concatenates subscripts with indexes into the variable names, with the number of the occurrence of that variable, similar to the Cryptol implementation

described in the previous section. The return value, is the last value of `nP`, which in this case, its nP_5.

As we can see, there are some optimizations that can be done, mostly because of the unnecessary memory that is being used when a store operation is made on a sequence. Also, the if-then-else conditionals except the line 49 may be also considered as dead code, i.e. if they are removed from the function it does not affect the program results.

6.4 Summary

In this chapter we have presented a brief introduction to elliptic curves for cryptography, focusing some aspects relevant to a practical implementation. The elliptic curves for cryptography were chosen as a case study essentially because of the smaller key sizes used by them, which makes them a primary choice for hardware design.

The first part of this chapter addresses the two languages and the implementation of elliptic curves. The data types used for the representation of elliptic curve points over the GF(2^{163}) field and the `Double` operation in CAO and Cryptol were presented. The imperative nature of the formal specification makes it easier to implement using the CAO language. However, the implementation in Cryptol is also possible, having some similarities.

Finally, it was presented the compilation of the `Double` algorithm from CAO to Cryptol, using the tool presented in the previous chapter. The results were positive, as it was possible to compile the output to VHDL using the Cryptol toolkit. However, the generated code can still be optimized, for example, by executing dead code elimination on it.

Chapter 7

Conclusions and future work

This chapter aims to present a summary of the work performed in the context of this dissertation, establishing some conclusions about the developed work, focusing on the contributions that were made. Additionally, a few considerations about future work on this subject will be made, discussing the state of the DSLs for cryptography and what features could be added to the CAO2Cryptol compiler.

7.1 Conclusions

This work presented two domain-specific languages for cryptography, Cryptol and CAO. The main purpose was to study these two languages and to understand how its features could be used to implement cryptographic algorithms.

- The Cryptol language is based on the functional paradigm, uses function composition to create programs, is stateless and has higher-order constructs such as polymorphism and other operators. The Cryptol toolkit is composed by a verification suite (that was not approached in this work) and a set of compiler tools to other languages such as C or VHDL.
- The CAO language, is an imperative language with a syntax similar to C's and aims to allow one to quickly re-target a cryptographic algorithm from its formal specification. It includes data types such as bit vectors or finite fields which are used widely in cryptography. Part of the CAO toolkit is also the CAO-SL tool that allows one to specify contracts in CAO functions, such as

pre-conditions and post-conditions. The CALF compiler, introduces higher-order features in the language such as polymorphism.

To compare the two languages, it was used the AES algorithm, which is an encryption scheme widely used nowadays to achieve information confidentiality. The implementation of the cipher of the algorithm in Cryptol and CAO was partially presented, essentially to compare the implementation of some parts of the algorithm in both languages. Most part of the differences were because of the distinct characteristics of the functional and imperative paradigms. On one hand, the functional approach uses function composition, recursion and sequence comprehension lists. On the other, the imperative implementation uses mostly iteration cycles such as `seq`'s and take advantage of the finite fields data type.

Furthermore, this work also studied the limitations of the Cryptol's compilation tools for hardware platforms. This was important to study the viability of building a CAO2Cryptol compiler. The variables of a CAO program are monomorphic and all values are known at compile time, which makes the compilation easier. However, a few transformations are required, mostly because the if-then-else statements in CAO, which can change the whole state of the program. In Cryptol, the if-then-else is an expression and must return the same type. The variables should be defined only once in a functional language, unlike imperative languages such as CAO. To solve this problem, the original CAO program is translated to the SSA form, which inserts indexes in the variables as subscripts. With this transformations, the CAO program is transformed into straight-line code which can be translated to Cryptol code using a pretty printer module, which writes Cryptol's syntax.

The main contributions from the work performed in this dissertation, besides the compilation, were mainly focused on the CAO language. The CFG and SSA components developed for the CAO language, may be incorporated in the future in the CAO compiler. In fact, the SSA form may be used to perform optimizations in CAO programs such as constant propagation or dead code elimination. The implementation of the elliptic curves and its algorithms was also a contribution, which can be used in the future as a library to be used by the CAO language.

7.2 Future work

The compilation tool developed in this dissertation used most of the features in the CAO language. However, the finite fields data type was not considered for the compilation tool. As this type is something that is used widely in cryptographic primitives and Cryptol does not support natively, would be an interesting addition to compiler. The *while* statement was not considered either, and could be added to the compiler also.

Because of the SSA transformation, the output code uses a lot of memory and that is notorious when container types such as vectors, matrices or structures are used: for each store operation, the whole container memory is rewritten in a new variable. If there are a huge number of store operations, the size of used memory would be high. Eventually, some optimizations could be done about this problem, specially when there are sequential store operations, which could be simplified into only one operation.

Most part of this work was part of a research grant inserted in the european SMART (Secure Memories and Applications-related Technologies) project, which aims to develop a new generation of portable and small smart secure devices, with focus on performance. A CAO2C compiler tool is being developed in the scope of this project, which can reuse components from the CAO2Cryptol compiler such as the CFG and the SSA form, in order to generate an optimized compiler.

Appendix A

T-box

Listing A.1: Calculation of the T_0-box table

```
T0_table : [256][32];
T0_table = [
0xa56363c6 0x847c7cf8 0x997777ee 0x8d7b7bf6 0x0df2f2ff 0xbd6b6bd6 0xb16f6fde 0x54c5c591
0x50303060 0x03010102 0xa96767ce 0x7d2b2b56 0x19fefee7 0x62d7d7b5 0xe6abab4d 0x9a7676ec
0x45caca8f 0x9d82821f 0x40c9c989 0x877d7dfa 0x15fafaef 0xeb5959b2 0xc947478e 0x0bf0f0fb
0xecadad41 0x67d4d4b3 0xfda2a25f 0xeaafaf45 0xbf9c9c23 0xf7a4a453 0x967272e4 0x5bc0c09b
0xc2b7b775 0x1cfdfde1 0xae93933d 0x6a26264c 0x5a36366c 0x413f3f7e 0x02f7f7f5 0x4fcccc83
0x5c343468 0xf4a5a551 0x34e5e5d1 0x08f1f1f9 0x937171e2 0x73d8d8ab 0x53313162 0x3f15152a
0x0c040408 0x52c7c795 0x65232346 0x5ec3c39d 0x28181830 0xa1969637 0x0f05050a 0xb59a9a2f
0x0907070e 0x36121224 0x9b80801b 0x3de2e2df 0x26ebebcd 0x6927274e 0xcdb2b27f 0x9f7575ea
0x1b090912 0x9e83831d 0x742c2c58 0x2e1a1a34 0x2d1b1b36 0xb26e6edc 0xee5a5ab4 0xfba0a05b
0xf65252a4 0x4d3b3b76 0x61d6d6b7 0xceb3b37d 0x7b292952 0x3ee3e3dd 0x712f2f5e 0x97848413
0xf55353a6 0x68d1d1b9 0x00000000 0x2cededc1 0x60202040 0x1ffcfce3 0xc8b1b179 0xed5b5bb6
0xbe6a6ad4 0x46cbcb8d 0xd9bebe67 0x4b393972 0xde4a4a94 0xd44c4c98 0xe85858b0 0x4acfcf85
0x6bd0d0bb 0x2aefefc5 0xe5aaaa4f 0x16fbfbed 0xc5434386 0xd74d4d9a 0x55333366 0x94858511
0xcf45458a 0x10f9f9e9 0x06020204 0x817f7ffe 0xf05050a0 0x443c3c78 0xba9f9f25 0xe3a8a84b
0xf35151a2 0xfea3a35d 0xc0404080 0x8a8f8f05 0xad92923f 0xbc9d9d21 0x48383870 0x04f5f5f1
0xdfbcbc63 0xc1b6b677 0x75dadaaf 0x63212142 0x30101020 0x1affffe5 0x0ef3f3fd 0x6dd2d2bf
0x4ccdcd81 0x140c0c18 0x35131326 0x2fececc3 0xe15f5fbe 0xa2979735 0xcc444488 0x3917172e
0x57c4c493 0xf2a7a755 0x827e7efc 0x473d3d7a 0xac6464c8 0xe75d5dba 0x2b191932 0x957373e6
0xa06060c0 0x98818119 0xd14f4f9e 0x7fdcdca3 0x66222244 0x7e2a2a54 0xab90903b 0x8388880b
0xca46468c 0x29eeeec7 0xd3b8b86b 0x3c141428 0x79dedea7 0xe25e5ebc 0x1d0b0b16 0x76dbdbad
0x3be0e0db 0x56323264 0x4e3a3a74 0x1e0a0a14 0xdb494992 0x0a06060c 0x6c242448 0xe45c5cb8
0x5dc2c29f 0x6ed3d3bd 0xefacac43 0xa66262c4 0xa8919139 0xa4959531 0x37e4e4d3 0x8b7979f2
0x32e7e7d5 0x43c8c88b 0x5937376e 0xb76d6dda 0x8c8d8d01 0x64d5d5b1 0xd24e4e9c 0xe0a9a949
0xb46c6cd8 0xfa5656ac 0x07f4f4f3 0x25eaeacf 0xaf6565ca 0x8e7a7af4 0xe9aeae47 0x18080810
0xd5baba6f 0x887878f0 0x6f25254a 0x722e2e5c 0x241c1c38 0xf1a6a657 0xc7b4b473 0x51c6c697
0x23e8e8cb 0x7cdddda1 0x9c7474e8 0x211f1f3e 0xdd4b4b96 0xdcbdbd61 0x868b8b0d 0x858a8a0f
0x907070e0 0x423e3e7c 0xc4b5b571 0xaa6666cc 0xd8484890 0x05030306 0x01f6f6f7 0x120e0e1c
0xa36161c2 0x5f35356a 0xf95757ae 0xd0b9b969 0x91868617 0x58c1c199 0x271d1d3a 0xb99e9e27
0x38e1e1d9 0x13f8f8eb 0xb398982b 0x33111122 0xbb6969d2 0x70d9d9a9 0x898e8e07 0xa7949433
0xb69b9b2d 0x221e1e3c 0x92878715 0x20e9e9c9 0x49cece87 0xff5555aa 0x78282850 0x7adfdfa5
0x8f8c8c03 0xf8a1a159 0x80898909 0x170d0d1a 0xdabfbf65 0x31e6e6d7 0xc6424284 0xb86868d0
0xc3414182 0xb0999929 0x772d2d5a 0x110f0f1e 0xcbb0b07b 0xfc5454a8 0xd6bbbb6d 0x3a16162c
    ];
```

Appendix B

CAO2Cryptol EC2Double function output

Listing B.1: EC2Double output using the CAO2Cryptol compilation tool

```
1  EC2Double : (GFPoint) -> GFPoint;
2  EC2Double (P) = nP_5
3          where {
4                  nP : GFPoint;
5                  nP = EC2infinity;
6                  c : [163];
7                  c = (...) // b^(2^(163-2))
8                  T1 : [163];
9                  T1 = P @ 0;
10                 T2 : [163];
11                 T2 = P @ 1;
12                 T3 : [163];
13                 T3 = P @ 2;
14                 T4 : [163];
15                 T4 = c;
16                 P_0, P_1 : GFPoint;
17                 P_0 = P;
18                 T1_0, T1_1, T1_2, T1_3, T1_4, T1_5 : [163];
19                 T1_0 = T1;
20                 T2_0, T2_1, T2_2, T2_3, T2_4, T2_5, T2_6 : [163];
21                 T2_0 = T2;
```

```
22          T3_0, T3_1, T3_2, T3_3 : [163];
23          T3_0 = T3;
24          T4_0, T4_1, T4_2, T4_3, T4_4, T4_5 : [163];
25          T4_0 = T4;
26          c_0 : [163];
27          c_0 = c;
28          nP_0, nP_1, nP_2, nP_3, nP_4, nP_5 : GFPoint;
29          nP_0 = nP;
30          nP_1 = EC2infinity;
31          T2_1 = mult(T2_0, T3_0);
32          T3_1 = mult(T3_0, T3_0);
33          T4_1 = mult(T3_1, T4_0);
34          T3_2 = mult(T1_0, T3_1);
35          T2_2 = sum(T2_1, T3_2);
36          T4_2 = sum(T1_0, T4_1);
37          T4_3 = mult(T4_2, T4_2);
38          T4_4 = mult(T4_3, T4_3);
39          T1_1 = mult(T1_0, T1_0);
40          T2_3 = sum(T1_1, T2_2);
41          T2_4 = mult(T2_3, T4_4);
42          T1_2 = mult(T1_1, T1_1);
43          T1_3 = mult(T1_2, T3_2);
44          T2_5 = sum(T1_3, T2_4);
45          T1_4 = T4_4;
46          nP_2 = take (0,nP_1) # [T1_4] # drop (0+1,nP_1);
47          nP_3 = take (1,nP_2) # [T2_5] # drop (1+1,nP_2);
48          nP_4 = take (2,nP_3) # [T4_4] # drop (2+1,nP_3);
49          nP_5 = if ((T1_0 == 0b0) | (T3_0 == 0b0)) then (nP_1) else (nP_4);
50          T4_5 = if ((T1_0 == 0b0) | (T3_0 == 0b0)) then (T4_0) else (T4_4);
51          T3_3 = if ((T1_0 == 0b0) | (T3_0 == 0b0)) then (T3_0) else (T3_2);
52          T2_6 = if ((T1_0 == 0b0) | (T3_0 == 0b0)) then (T2_0) else (T2_5);
53          T1_5 = if ((T1_0 == 0b0) | (T3_0 == 0b0)) then (T1_0) else (T1_4);
54          P_1 = if ((T1_0 == 0b0) | (T3_0 == 0b0)) then (P_0) else (P_0);
55      };
```

Bibliography

[AES01] Specification for the advanced encryption standard (aes). Federal Information Processing Standards Publication 197, 2001.

[All70] F.E. Allen. Control flow analysis. In *ACM Sigplan Notices*, volume 5, pages 1–19. ACM, 1970.

[AP02] A.W. Appel and J. Palsberg. *Modern compiler implementation in Java*. Cambridge Univ Pr, 2002.

[BAC+09] Manuel Barbosa, José Almeida, Alcino Cunha, , Andrew Moss, Dan Page, Jorge Pinto, Bárbara Vieira, and Nuno Rodrigues. Formal specification language definitions and security policy extensions. Technical Report D5.2, CACE Project Deliverable, 2009.

[BMP+11] M. Barbosa, A. Moss, D. Page, N.F. Rodrigues, and P.F. Silva. Type checking cryptography implementations. Technical report, Technical Report DI-CCTC-11-01, CCTC, Univ. Minho, 2011.

[CFR+91] R. Cytron, J. Ferrante, B.K. Rosen, M.N. Wegman, and F.K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(4):451–490, 1991.

[CHK01] K.D. Cooper, T.J. Harvey, and K. Kennedy. A simple, fast dominance algorithm. *Software Practice and Experience*, 4:1–10, 2001.

[cry08] Cryptol programming guide. Technical report, Galois, Inc., Portland, Oregon., 2008.

[DR99] J. Daemen and V. Rijmen. AES proposal: Rijndael. 1999.

[DR02] Joan Daemen and Vincent Rijmen. *The design of Rijndael: AES — the Advanced Encryption Standard*. Springer-Verlag, 2002.

[EKP98] P. Eles, K. Kuchcinski, and Z. Peng. *System synthesis with VHDL*, volume 135. Kluwer Academic Publishers, 1998.

[Fow10] M. Fowler. *Domain specific languages*. 2010.

[HLHM00] D. Hankerson, J. López Hernandez, and A. Menezes. Software implementation of elliptic curve cryptography over binary fields. In *Cryptographic Hardware and Embedded Systems—CHES 2000*, pages 243–267. Springer, 2000.

[Huf10] T. Huffmire. Handbook of fpga design security. 2010.

[IEE99] IEEE. Ieee p1363 / d13 (draft version 13) - standard specifications for public key cryptography. 1999.

[Inc08] Galois Inc. Cryptol tutorial. Technical report, Galois, Inc., Portland, Oregon., 2008.

[Kob87] N. Koblitz. Elliptic curve cryptosystems. *Mathematics of computation*, 48(177):203–209, 1987.

[LLS09] H. Lee, K. Lee, and Y. Shin. Aes implementation and performance evaluation on 8-bit microcontrollers. *Arxiv preprint arXiv:0911.0482*, 2009.

[Mil86] V. Miller. Use of elliptic curves in cryptography. In *Advances in Cryptology—CRYPTO'85 Proceedings*, pages 417–426. Springer, 1986.

[Sch93] B. Schneier. *Applied cryptography: protocols, algorithms, and source code in C*. John Wiley & Sons, Inc., 1993.

[VOMV96] P.C. Van Oorschot, A.J. Menezes, and S.A. Vanstone. *Handbook of applied cryptography*. Crc Press, 1996.

[WFH90] D.A. Watt, W. Findlay, and J. Hughes. *Programming language concepts and paradigms*, volume 234. Prentice Hall, 1990.