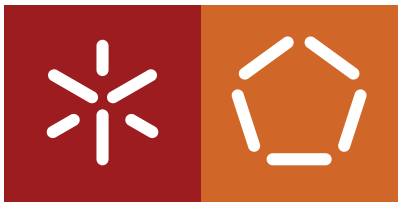
Universidade do Minho
Escola de Engenharia
Departamento de Informática

Vitor Manuel Parreira Pereira

A deductive verification tool for cryptographic software

January 2016



Universidade do Minho
Escola de Engenharia
Departamento de Informática

Vitor Manuel Parreira Pereira

A deductive verification tool for cryptographic software

Master dissertation
Master Degree in Computing Engineering

Dissertation supervised by
Manuel Bernardo Martins Barbosa
José Carlos Bacelar Almeida

January 2016

ACKNOWLEDGMENTS

Two years have past since I enrolled my masters program. In the end, it all comes to this document: this is the document that will be rated and that will define my final classification. But for me, this thesis is much more than the realisation of a project. It is the culmination of two great years, that were influenced by a very special set of people, to whom I am grateful for.

Primarily, I would like to thank my supervisors, Prof. Manuel Bernardo Martins Barbosa and Prof. José Carlos Bacelar Almeida. I cannot imagine a better duo to supervise me! To Prof. Manuel Barbosa, for all the patience, all the joy, all the guidance and all the opportunities that you provided me. I really enjoyed our collaboration and I hope it lasts many years! To Prof. Bacelar, for all the important and wise comments that helped to unlock many difficulties found in the project. Thank you, thank you both very much!

I would like to thank the person that wakes me up every day with a hearty ”Good morning!”, my dear girlfriend Filipa Cruz. Even though I could not count on you physically, you were always with me and available for every thing that I wanted. You have the ability to cheer me up every time and your presence is enough to make me win the day. We were made for helping, not for ruling, made for love.

To all the *MFES* family, thank you! Thank you for your help integrating me in the university and in the city. I cannot finish this paragraph without mentioning four people. To Paulo Silva, my dear Paulo Silva, thank you for your energy, for your unconditional friendship and for many joyful moments you gave me! I will never, but never, forget that lunch, where I heard your loud voice yelling ”Ó patrão, senta aqui com a gente!”. If there are seconds that can change a lifetime, those were the seconds. To Óscar Pereira, the Indian that is not Indian, thank you for your patience. I cannot imagine how many patience is needed to endure me! I’m glad that we will continue to work together, side by side, at least another year. To Victor Cacciari Miraldo, thank you for the most crazy moments you provided me and the group and for all the nights spent in my couch, developing the most exoteric projects. It is a pleasure to receive you every time you visit Braga and I hope you have a couch for me in the Netherlands! Finally, to Jorge Lobo. You, literally, blasted my mind. I think in a different way, I see in a different way and I live in a different way thanks to you. Thank you for your unlimited wisdom and intelligence, for the most crazy conversations we had and for the friendship we developed. I don’t know about you, but when I met you, I felt that you were friends for 10 years already!

I cannot forget those that are not with me right now, but that will never stop to be my friends. Tiago Carvalho, Diogo Oliveira and Nuno Garcia, no one will ever replace you and wipe you out of my memory. Tiago Carvalho, the laziest person I know, you may be quiet and shy, but you are there every time I need you. For that, thank you. To Diogo Oliveira, thank you for 17 years of friendship. It is

interesting to see how our friendship evolved. You should try this mentar exercise some time! And, to Nuno Garcia, thank you for being with me in the most important moments of my life in Covilhã. It wouldn't be the same without you. I sincerely hope that, somewhere in the future, we will be reunited once more my friends!

I also thank to my lifetime friends Bruno Costa, Joana Diniz and Inês Martins. I know I can count on you every time and that you will always support me, no mather how. Thank you all, for everything!

A special thank to all my family, that, one way or another, kept side by side with me during these two years. Of course I need to individualise my dear cousin Xana, for being one of the most important people of my life and for being my life mate!

A special gratitude to my godfather Paulo Fiadeiro, for being present in the good and in the bad moments. It is not easy to express with words the thanks you deserve. I will be forever grateful for what you have done for me and it is a privilege to be your friend!

To the person that will, for sure, write his name in the history of Computer Science, my brother Mário Pereira, I leave here my great appreciation. Thank you for your availability, whenever I need, no matter the distance. Keep it up, you will make history!

To my mother Ernesta Parreira, who does not know English, I still write her a paragraph. Of course I do, it was impossible not to thank for supporting me in every new journey of my life, no matter how much it will cost you. I will be there for you every time you need, because from now on, it is me who will take care of you!

Finally, my last gratefulness belongs to my father Mário Pereira. I'm certain that you are reading this letters I am writing and you are smilling on then, just like you did on everything. I'm sure that all the strength I feel comes directly from you and that you will continue to be on my side every day. Many thanks to you dad, for all that you though me.

ABSTRACT

Security is notoriously diffcult to sell as a feature in software products. In addition to meeting a set of security requirements, cryptographic software has to be cheap, fast, and use little resources. The development of cryptographic software is an area with specific needs in terms of software development processes and tools. In this thesis we explore how formal techniques, namely deductive verification techniques, can be used to increase the guarantees that cryptographic software implementations indeed work as prescribed.

CAO (C and OCCAM) is a programming language specific to the domain of Cryptography. Control structures are similar to C, but it incorporates data types that deal directly with the needs of a programmer when translating specifications of cryptographic schemes (eg, from scientific papers or standards) to the real world. CAO language is supported by a compiler and an interpreter developed by HASLab, in a sequence of research and development projects.

The CAOVerif tool was designed to allow deductive verification programs written in CAO. This tool follows the same paradigm as other tools available for high level programming languages, such as Frama-C, according to which a CAO program annotated with a specification is converted in an input program to the Jessie/Why3 tool-chain, where the specified properties are then analysed.

After the development of CAOVerif, a new tool, specific to the domain of Cryptography - named EasyCrypt - was developed. The objective of this project is to evaluate EasyCrypt as a potential backend for the CAOVerif tool, through the development of a prototype that demonstrates the advantages and disadvantages of this solution.

RESUMO

O software criptográfico possui requisitos específicos para garantir a segurança da informação que manipula. Além disso, este tipo de software necessita de ser barato, rápido e utilizar um número reduzido de recursos. Garantir a segurança da informação que é manipulada por tais sistemas é um grande desafio, sendo por isso de grande objecto de estudo actualmente. Nesta tese exploramos como as técnicas formais, nomeadamente as técnicas de verificação dedutiva, podem ser utilizadas por forma a garantir que as implementações de *software* criptográfico funcionam, de facto, como prescrito.

CAO (C and OCCAM) é uma linguagem de programação específica para o domínio da criptografia. As estruturas de controlo são semelhantes às da linguagem C, mas incorpora tipos de dados e operadores que tratam de forma direta as necessidades de um programador na transposição de especificações de esquemas criptográficos (e.g., de artigos científicos ou standards) para o mundo real. A linguagem CAO é suportada por um compilador e por um interpretador desenvolvidos pelo HASLab numa sequência de projetos de investigação e desenvolvimento.

A ferramenta CAOVerif foi concebida para permitir a verificação dedutiva de programas escritos em CAO. Esta ferramenta segue o paradigma de outras ferramentas disponíveis para linguagens de programação de alto nível, como por exemplo o Frama-C, de acordo com o qual um programa CAO anotado com uma especificação é convertido num programa de *input* para a *tool-chain* Jessie/Why3, onde as propriedades especificadas são depois analisadas.

Depois do desenvolvimento do CAOVerif surgiu uma nova ferramenta de verificação específica para o domínio da criptografia, denominada EasyCrypt. O objetivo desta proposta de dissertação é a avaliação do EasyCrypt como potencial *backend* da ferramenta CAOVerif, através do desenvolvimento que um protótipo de demonstre as vantagens e desvantagens desta solução.

CONTENTS

LIST OF FIGURES

List of Figures

1

INTRODUCTION

Cryptography, as the use of codes and ciphers to protect messages, began thousands of years ago. In fact, one can find registers of the use of cryptography since the Old Kingdom of Egypt - where hieroglyphs were used to obfuscate data - the ancient Greece - that used the scytale transposition cipher in the Spartan military - or even in the Roman Empire - where the Caesar cipher and its derivations were widely used.

However, the cryptographic techniques mentioned above were monoalphabetical, i.e., a key of the form $\mathcal{A} \to \mathcal{A}$ (for some alphabet $\mathcal{A}$) was fixed and every character of the input message was mapped to some other character according to the substitution key. These cryptosystems were easily broken with the invention of frequency analysis attacks. A frequency analysis attack consists in performing an analysis on the number of times that each letter of the alphabet appears in the ciphertext and then using a frequency table - containing the letters of $\mathcal{A}$ and the frequency with which each letter appears in a common text written in that alphabet - to invert the substitution made during the encryption.

Therefore, cryptography advanced to the polyalphabetical ciphers. The main difference between these encryption techniques and the previous ones is the key: instead of using a simple substitution key $\mathcal{A} \to \mathcal{A}$, keys of the form $\mathcal{A} \to \mathcal{A}^*$ started to be used and the one letter from the alphabet $\mathcal{A}$ could be mapped into any possible letter of $\mathcal{A}$ instead of just one. Nevertheless, these cryptosystems were, one more time, broken with recurse to frequency analysis attacks.

It was until the XX century that cryptography suffered more major changes, with the invention of the one-time pad. The one-time pad consists in using one key $k \in \{0,1\}^*$, with the same size of the message $m \in \{0,1\}^*$ and perform the bitwise XOR operation in order to get a ciphertext $c \in \{0,1\}^*$. This is the only cryptosystem that can not be broken, if used correctly (one key is used to cipher one and only one message). Yet, one-time pad has a lot of overheads that makes it impossible to use it in practice, like the need for the key to be of the same size of the plaintext, the need to generate a new key every time in order to cipher a new message and even the fact that it is only secure with respect to passive adversaries.

During WWII, mechanical and electromechanical cipher machines were in wide use, as the Germans made heavy use of the well known Enigma machine. The Enigma machine was only broken with recurse to reverse engineering that was made using mathematical methods. This event created a major revolution in cryptography, which stopped to be a subject of linguistics sciences to become a field of study in mathematics. This fact led to the creation of modern cryptography, where mathe-

matics assume a major importance in the construction of cryptosystems. The security of encryption schemes started to be reduced to some mathematical assumptions that are believed to be hard to solve in the existing computational context.

Nowadays, cryptography allows the construction of the most wide range of primitives. One can use cryptographic primitives to ensure confidentiality over data, its integrity, authenticity, privacy, etc. And with the proliferation of technology in modern society, cryptography started to be incorporated in all possible devices, as there is the need to associate security properties to almost all computations that are made.

The development of cryptographic software is clearly distinct from other areas of software engineering. The design and implementation of cryptographic software draws on skills from mathematics, computer science and electrical engineering. Also, since security is difficult to sell as a feature in software products, cryptography needs to be as close to invisible as possible in terms of computational and communication load. As a result, cryptographic software must be optimised aggressively, without modifying the security semantics. Finally, cryptographic software is implemented on a very wide range of devices, from embedded processors with very limited computational power and memory, to high-end servers, which demand high-performance and low-latency. Therefore, the implementation of cryptographic kernels imposes a specific set of challenges that do not apply to other system components. For example, direct implementation in assembly language is common, not only to guarantee a more efficient implementation, but also to ensure that low-level security policies are satisfied by the machine code.

1.1 THE CAO LANGUAGE

The CAO language Barbosa et al. (2012) aims to change this state of affairs, allowing natural description of cryptographic software implementations, which can be analysed by a compiler that performs security-aware analysis, transformation and optimisation. The driving principle behind the design of CAO is that the language should support cryptographic concepts as first-class language features. Unlike the languages used in mathematical software packages such as Magma or Maple, which allow the description of high-level mathematical constructions in their full generality, CAO is restricted to enabling the implementation of cryptographic components such as block ciphers, hash functions and sequences of finite field arithmetic for Elliptic Curve Cryptography (ECC).

CAO preserves some higher-level features to be familiar to an imperative programmer, whilst focusing on the implementation aspects that are most critical for security and efficiency. The memory model of CAO is, by design, extremely simple to prevent memory management errors (there is no dynamic memory allocation and it has call-by-value semantics). Furthermore, the language does not support any input/output constructions, as it is targeted at implementing the core components in cryptographic libraries. In fact, a typical CAO program comprises only the definition of a global state and a set of functions that allow performing cryptographic operations over that state. Conversely, the

native types and operators in the language are highly expressive and tuned to the specific domain of cryptography. In short, the design of CAO allowed trading off the generality of a language such as C or Java, for a richer type system that permits expressing cryptographic software implementations in a more natural way.

CAO introduces as first-class features pure incarnations of mathematical types commonly used in cryptography (arbitrary precision integers, ring of residue classes modulo an integer, finite field of residue classes modulo a prime, finite field extensions and matrices of these mathematical types) and also bit strings of known finite size. A more expressive type system would be expected from any domain-specific language. However, in the case of CAO, the design of the type system was taken a step further in order not only to allow an elegant formalisation of the type checking rules, but also to allow the efficient implementation of a type checking system that performs extensive preliminary validation of the code, and extracts a very rich body of information from it. This fact makes the CAO type checker a critical building block in the implementation of compilation and formal verification tools supporting the language.

1.2 DEDUCTIVE PROGRAM VERIFICATION

Program verification is the area of Formal Methods that aims to statically check software properties based on the axiomatic semantics of programming languages. In this master thesis we focus on techniques based on Hoare logic, brought to practice through the use of contracts – specifications consisting of preconditions and postconditions, annotated into the programs. Verification tools based on contracts are becoming increasingly popular, as their scope evolved from toy languages to realistic fragments of languages like C, C#, or Java. We use the expression deductive verification to distinguish this approach from other ways of checking properties of programs, such as model checking.

1.3 VERIFICATION OF CRYPTOGRAPHIC SOFTWARE

One may be tempted to define the verification of cryptographic software as a subset of software verification and use deductive program verification in order to perform the desired proofs. However, cryptographic software is not deterministic (as there are, for example, random samplings in key generation algorithms or in encryption algorithms) and the Hoare logic does not deal with these probabilistic states. Therefore, there is the need to extend the Hoare logic with means to perform probabilistic reasoning, resulting in the probabilistic Hoare logic.

Additionally, when dealing with cryptography, developing functional correctness or safety proofs is important but one must also prove that some scheme provides the desired security property. Typical security proofs consist in reducing the security of some primitive to some mathematical assumption that is believed hard in the existing computation model. At the end of the reduction, one may argue

that breaking the security property of the primitive is as hard as breaking the mathematical assumption and that there are no polynomial time adversaries that are able to do so.

A reduction proof can be made using a game-based approach: one defines a cryptographic game (a program where an adversary tries to break a security property), that represents the advantage that an adversary has in breaking the security property of some given primitive. After, one can build a sequence of indistinguishable games and prove the equivalence between all the games in the proof. Obviously, one needs to have some form of reasoning about equivalences between programs. This is done using probabilistic relational Hoare logic, which contemplates a calculus that allows the comparison between two probabilistic programs. If an equivalence proof between two programs C and C' is successful, then C and C' are computationally indistinguishable.

1.4 EASYCRYPT

EasyCrypt is a toolset for reasoning about relational properties of probabilistic computations with adversarial code. Its main application is the construction and verification of game-based cryptographic proofs. Initial applications of EasyCrypt focus on encryption and signature schemes.

EasyCrypt, as an interactive proof assistant, contemplates an ambient logic to deal with propositions, the original Hoare logic to deal with deterministic reasoning, the probabilistic Hoare logic to allow one to reason about some probabilistic state and the probabilistic relational Hoare logic in order to perform equivalence proofs.

The EasyCrypt tool incorporates some mechanisms that allow a better structuration of the proof, like *theories* and *modules*. A *theory* allows the high-level description of some mathematical structure, that can after be instantiated to some concrete implementation. For example, one can define a *Monoid theory*, stating that a monoid is some algebraic structure with elements of some type τ, with an operation $\oplus$ and with two axioms associated with τ and $\oplus$ - associativity of $\oplus$ ($\forall x, y, z : \tau, (x \oplus y) \oplus z = x \oplus (y \oplus z)$) and the existence of an identity element ($\exists y : \tau, \forall x : \tau, x \oplus y = x$). After, the monoid structure can be instantiated with a concrete type. For example, one can instantiate it with type $\tau = bool$ and with a concrete operation XOR, which verifies the previous axioms.

A *module* is a syntactic unit that allows the declaration of programs (procedures) and variables. Later, it is possible to reason about properties over these programs using Hoare logic, probabilistic Hoare logic or probabilistic relational Hoare logic, combined with the available reasoning tactics that EasyCrypt incorporates.

Although it seems that EasyCrypt is limited to the scope of cryptography and to the development of security proofs, it allows a lot of different reasonings. Using EasyCrypt's simple while language language, one can easily write programs and reason about their functional correctness or even their safety. This is done by building Hoare triples and, using the *Verification Condition Generator* (VC-Gen) that is embedded in EasyCrypt, creating a proof tree can be discharged using SMT solvers, through the Why3 tool.

1.5 MOTIVATION

We have seen that cryptographic software is particularly different from other kinds of software and that it interleaves a lot of distinct areas in its development. Consequently, its correct development can turn out to be a very arduous process. Plus, it is also of great importance to be able to perform all the necessary proofs around cryptographic software.

We have also seen that one can overcome the first identified problem using the CAO language and overcome the second one using EasyCrypt. However, there is no way of connecting the two platforms and, in some way, being able to write cryptographic code using a language close to standards and perform proofs about it in a tool suited to cryptographic reasoning.

The project of this master thesis, and its subsequent tool, aims to connect the two *worlds* presented in the previous paragraph. We present a translation between CAO and EasyCrypt that will permit to write the desired code in a friendly language, annotate it, and then to be able to map their code into an EasyCrypt script, where one can perform functional correctness proofs, safety proofs and security proofs. After the proofs being completed, one can generate very efficient C code, that corresponds to the CAO implementation.

1.6 OBJECTIVES

The main objective of this master thesis is to develop a deductive verification platform for CAO programs, using EasyCrypt as a backend for the tool. The main idea is to map an annotated CAO program to an EasyCrypt script, and be able to automatically generate all the proof scripts that are necessary to prove functional correctness and safety of CAO programs.

More in detail, the objectives of this master thesis are the following:

- Investigate the state of the art of deductive verification of cryptographic software - studying the different approaches that exist to deductive verification of cryptographic software, in order know how to develop a tool that takes all the advantages of the existing tools but that try to overcome their difficulties.
- Get involved with the CAO platform and with the EasyCrypt tool - since CAO and EasyCrypt are two essential components of the project, it represents an important objective to gain proficiency with the two tools. Particularly, it is important to understand the semantics of CAO programs, how to derive EasyCrypt scripts from CAO code and to understand the proof mechanisms of EasyCrypt.
- Investigate how EasyCrypt can be used to prove properties about CAO programs - EasyCrypt was originally developed to support reasoning about probabilistic programs, with particular focus in cryptography. Therefore, it is important to know how to use EasyCrypt to develop functional correctness proofs and safety proofs.

- Develop a mapping algorithm between a CAO program and an EasyCrypt scheme - the mapping between CAO programs and EasyCrypt scripts is of great importance for this work. This mapping algorithm needs to be sound, so that one is able to fully trust the tool.
- Validate the developed prototype - in order to ensure the correctness and validity of the developed tool, it will be developed a battery of test.
- Evaluate the use of EasyCrypt as backend for the CAOVerif tool - the CAOVerif tool was developed before the existence of EasyCrypt and makes use of Frama-C Baudin et al. (2010) to perform deductive verification of CAO programs. However, despite being a friendly platform to use, it has some overheads in what concerns time and space efficiency, which can be completely overcome if one uses EasyCrypt as a backend instead. The final goal of this thesis is to do a benchmark analysis of the CAOVerif tool, using Frama-C as backend and using EasyCrypt instead.

1.7 DOCUMENT STRUCTURE

In the second chapter we introduce some theoretical background that form the basis of our contributions. In particular, we follow the pipeline of formal verification of software, starting by presenting some concepts about type systems - the first step of the verification of software - and then revisiting the notions of program reasoning through the presentation of the Hoare logic, used to reason about functional properties of programs. We show two extensions of the Hoare logic - the probabilistic Hoare logic and the probabilistic relational Hoare logic - that are more suitable to the work developed. We end the chapter by introducing more specific concepts about software formal verification that were followed in this thesis and by presenting some state of the art related to the objects of study of the project.

In chapter three we present the CAO language, that was one central part of this project. We show the formal syntax of the language and then present its type system. In this project, the CAO language was extended with additional features and a deductive verification tool for the verification of programs written in CAO was developed.

Chapter four focuses on the CAO-SL language, which is the specification language intended to be used in annotations of CAO programs. Annotations have a central role in deductive verification of software: together with program instructions, annotations allow one to prove functional properties about a program. We present the possible annotations allowed by CAO-SL, as well as a formal grammar for them.

Chapter five is concerned to the presentation of the EasyCrypt toolset. EasyCrypt is a toolset for reasoning about relational properties of probabilistic computations with adversarial code. Its main application is the construction and verification of game-based cryptographic proofs. EasyCrypt was

used as a backend for the deductive verification tool for CAO. In this chapter, we introduce the basic EasyCrypt proof mechanisms, as well a proof example.

In chapter six we present the main contributions of this thesis. We start by showing a new implementation of the CAO typechecker, resulting in an unified language that contemplates the CAO language, extended with more features and syntactic domains, and the CAO-SL language. Next, we show our formalisation of the CAO type system in EasyCrypt and we present our translation algorithm, that maps an annotated CAO program into an EasyCrypt script. We end the chapter with an explanation of how the tool deals with safety properties of CAO programs.

Finally, in the last chapter of the thesis we summarise our conclusions of the work done, review the objectives initially proposed for this work and outlook some possible lines of future work.

2

THEORETICAL BACKGROUND

2.1 TYPE SYSTEMS

Type systems are introduced in programming languages with the aim of providing a level of static checking, with a *type* seen as a formal and concise description of the behaviour of some term (program). The main purpose of a type system is to define interfaces between different parts of a computer program and then check if the parts have been connected in a consistent way. Informally, a type system assigns a type to some term (for example, a variable or a function) and then checks if, accordingly to that type, the term is being used in a correct way. For example, given two variables x and y of the type Int and the addition operation over the integers $+$, a new term can be formed combining the two variables as $x + y$ because both variables are of the correct type. The process of verifying if some program is written in accordance to its type system is called *typechecking*. This checking can happen statically (at compile time), dynamically (at run time), or it can happen as a combination of static and dynamic checking.

Type systems are often specified as part of programming languages and built into the interpreters and compilers for them, in order to prevent *type errors* (a certain kind of value being used with values for which some operation does not make sense) or, for some programming languages, to prevent *memory errors* (invalid accesses to some value in the memory of a computer).

In what follows this section, we provide a lightweight explanation about type systems, mainly focused on its *syntax*, following the work of Luca Cardelli in Cardelli (1997).

2.1.1 *Judgements*

The description of the type system of a programming language is analogous to the description of the programming language syntax using a formal grammar, since a type system specifies the type rules of a programming language independently of particular typechecking algorithms.

Type systems are describe by a particular formalism called *judgements*, with the form

$$\Gamma \vdash \Phi$$

where Φ is an assertion and Γ is a *type environment*. A type environment is a structure that maps distinct variables to their types, usually defined as a map. The collection of variables declared in Γ is defined as $dom(\Gamma)$, i.e., the domain of Γ. All free variables in the assertion Φ must be declared in Γ.

A *typing judgement* is used to assert that a term M has a type τ with respect to some environment Γ. It has the form

$$\Gamma \vdash M :: \tau$$

meaning that M has type τ in Γ. For example, $[x :: Int] \vdash x + 1 :: Int$ means that the term $x + 1$ has type Int, provided that x has type Int.

One is able to reason about any judgement, by regarding it as valid or invalid, therefore formalising the notion of well typed programs. In order to distinguish between valid and invalid judgements, we introduce the notion of *type rules*, which allow a more suitable way for stating and proving technical lemmas and theorems over type systems.

2.1.2 *Type rules*

A type rule is composed of judgements and it aims to assert the validity of some judgements on the basis of other judgements that are assumed to be valid and has the following form

$$\frac{\Gamma_1 \vdash \Phi_1 \ ... \ \Gamma_n \vdash \Phi_n}{\Gamma \vdash \Phi}$$

meaning that if the upper judgements ($\Gamma_1 \vdash \Phi_1 \ ... \ \Gamma_n \vdash \Phi_n$) - called *premises* - hold, then the lower judgement ($\Gamma \vdash \Phi$) - called *conclusion* - also holds.

Recall the previous example. Suppose two given variables, x and y, that are of type Int in some well-formed environment Γ. The two variables can be combined to form a larger expression $x + y$, which also has type Int. The following type rule describes this behaviour.

$$\frac{\Gamma \vdash x :: Int \quad \Gamma \vdash y :: Int}{\Gamma \vdash x + y :: Int}$$

A collection of type rules is called a *(formal) type system*. Technically, type systems fit into the general framework of formal proof systems: collections of rules used to carry out step-by-step deductions. The deductions carried out in type systems concern the typing of programs.

2.1.3 *Type derivations*

A derivation is a tree of judgments with leaves at the top and the root at the bottom, where each judgement is obtained from the one immediately above it by some rule. In a type system, a derivation aims to prove that a root - containing a type judgement for a term - is valid and that the term is well

formed in what respect to types. Naturally, a fundamental requirement on type systems is that it must be possible to check whether or not a derivation is properly constructed, otherwise one would not be able to perform any typechecking on any language.

A valid judgement is one that can be obtained by building a correct derivation in a given type system, i.e., by correctly applying type rules. An example follows.

$$\dfrac{\dfrac{\dfrac{}{\emptyset \vdash \diamond}}{\emptyset \vdash 1 :: Int} \quad \dfrac{\dfrac{}{\emptyset \vdash \diamond}}{\emptyset \vdash 2 :: Int}}{\emptyset \vdash 1 + 2 :: Int}$$

In the presented derivation, the objective is to attest that the term $1 + 2$ has type Int in the empty environment $\emptyset$. Thus, to build the derivation, one starts by dividing the bottom judgment into two new judgments, one for each term. The literal terms 1 and 2 have type Int in the empty environment and the derivation is concluded by the following fundamental rule that states that the empty environment is always well formed.

$$\dfrac{}{\emptyset \vdash \diamond}$$

2.1.4 *Well typing and type inference*

Informally, a term M is well typed if it can be given some type. Formally, this statement means that, in an environment Γ, there is a type τ such that $\Gamma \vdash M :: \tau$ is a valid judgment.

The process of finding a derivation and a type for some term is called *type inference*. For the example above, a type could be inferred for the term $1 + 2$ because it was possible to build a derivation tree for it. However, suppose there exists another judgement $\Gamma \vdash \mathit{true} :: \mathit{Bool}$. A type could not be inferred for the term $1 + \mathit{true}$ because it was not possible to build a derivation for the term. We say that $1 + \mathit{true}$ is not typeable. Nevertheless, if one adds a the rule

$$\dfrac{\Gamma \vdash M :: Int \quad \Gamma \vdash N :: \mathit{Bool}}{\Gamma \vdash M + N :: Int}$$

, the previous term would be typeable. Hence, the type inference process is very sensitive to the type system in question.

2.2 WHILE LANGUAGE

In this section, we describe the general structure of the simple imperative languages, which are usually called *While* languages. These languages only contemplate simple constructions, such as the *skip* command (do nothing command), assignments, sequential composition, while loops and conditional executions. A more detailed syntax is of the *While* language is shown in Figure 1.

$$
\begin{array}{lcl}
\text{Exp}_{\tau} \ni e^{\tau} & :: & x^{\tau} \\
& | & f^{t}(e_1^{\tau_1}, ..., e_n^{\tau_n}), \text{ for } f^{\tau} \in \mathcal{F}^{Exp}, \text{ with } \tau \neq bool \text{ and } \text{rank}(f^{\tau}) = \langle \tau_1, ..., \tau_n \rangle \\
\text{Exp}_{bool} \ni b & :: & \textbf{true} \\
& | & \textbf{false} \\
& | & \neg b \\
& | & b_1 \wedge b_2 \\
& | & b_1 \vee b_2 \\
& | & e_1 = e_2 \\
& | & e_1 \neq e_2 \\
& | & e_1 ¡ e_2 \\
& | & e_1 \leq e_2 \\
& | & e_1 ¿ e_2 \\
& | & e_1 \geq e_2 \\
\text{Comm} \ni C & :: & \textbf{skip} \\
& | & C_1 \,;\, C_2 \\
& | & x^{\tau} := e^{\tau} \\
& | & \textbf{if } b \textbf{ then } C_1 \textbf{ else } C_2 \\
& | & \textbf{while } b \textbf{ do } C
\end{array}
$$

Figure 1: *While* language syntax.

We abstract the types of the *While* language since it can be instantiated for all types. For example, one can define a $While^{int}$ language, which represents a *While* language that enables integer operations or a $While^{int\ array}$ language, that contemplates operations over integers and integer arrays.

Despite being a very simple language, the *While* language is of a great interest for Computer Science, mainly in the scope of deductive program verification, because one can define a set of rules and axioms around it to be able to prove properties like functional correctness or safety. The derivation rules are of the form

$$\frac{\theta}{\{\phi\}C\{\psi\}}$$

where $C \in Comm$, θ, ϕ, ψ are first-order logic predicates and the objective it to prove that, if θ holds, then if ϕ holds at some state and C is executed in that state, ψ will hold at the resulting state of the execution of C.

2.3 HOARE LOGIC

The Hoare Logic Hoare (1969) is a formal system to reason about correctness of imperative programs, building on first-order logic. For now, we will study Hoare logic for simple *While* programs.

The notion of correctness in Hoare Logic is expressed in terms of Hoare triples used to specify the desired behaviour of the underlying programs. This specification consists of a precondition and a postcondition. The correctness of a program with respect to a given specification is asserted by

constructing a derivation in the inference system of Hoare logic that attests the validity of the Hoare triple. While doing so, one must identify an invariant for every loop in the program.

In what follows this chapter, we will introduce some theoretical concepts inherent to the formal verification of software, having as reference the book Almeida et al. (2011).

2.3.1 *Annotated While language*

In order to able to correctly reason about programs written in *While* language, there is the need to extend its syntax with the introduction of means to add annotations, i.e., pre- and postconditions to functions, loop invariants and possible assertions in the middle of the code. Preconditions are predicates that hold at the beginning of the execution of a piece of code and, inversely, postconditions are predicates that hold at the end of the execution of a piece of code. A loop invariant is any property that is preserved by executions of the loop's body. Additionally, one may also provide a loop variant, that represents some value that is used to prove the termination of the loop. These annotations are typically introduced at the definition of a function, meaning that, given a function F, annotated with the precondition ϕ and with the postcondition ψ - $\{\phi\}F\{\psi\}$ -, if ϕ is assumed to hold at the beginning of the execution of F, ψ will hold at the end of the execution of F.

The *While* language is then extend as presented in Figure 2.

$$
\begin{array}{rcl}
\text{Comm} \ni C & :: & \ldots \\
& | & \textbf{while } b \textbf{ do } \theta\ C \\
& & \ldots \\
\text{Assert} \ni \phi, \psi, \theta & :: & t^{bool} \\
& | & \textbf{true} \\
& | & \textbf{false} \\
& | & \neg\phi \\
& | & \phi \wedge \psi \\
& | & \phi \vee \psi \\
& | & \phi \Rightarrow \psi \\
& | & \phi \Leftrightarrow \psi \\
& | & \forall x^{\tau}.\phi \\
& | & \exists x^{\tau}.\phi
\end{array}
$$

Figure 2: Extension to the *While* language - annotations

2.3.2 *Specifications and Hoare triples*

The correction of a program is defined relatively to the specification of that program. The specification of a program consists in a precondition - an assertion that is assumed to hold at the beginning of the

execution of the program - and a postcondition - an assertion that must hold at the end of the execution of that program.

The notion of correctness in Hoare logic can be expressed in two ways:

- Total correctness - given a precondition ϕ, a postcondition ψ and some program C, such that ϕ and ψ are specifications for C, if ϕ holds in a given state and C is executed in that state, then the execution of C will stop, and moreover ψ will hold in the final state of execution;
- Partial correctness - given a precondition ϕ, a postcondition ψ and some program C, such that ϕ and ψ are specifications for C, if ϕ holds in a given state and C is executed in that state, then either execution of C does not stop, or if it does, ψ will hold in the final state;

To be able to reason formally about correctness, we introduce two new classes of formulas called Hoare triples.

$$[\phi]C[\psi] \mid \{\phi\}C\{\psi\}$$

These formulas correspond to partial and total correctness respectively: the Hoare triple $[\phi]C[\psi]$ is valid when the program C is totally correct with respect to the precondition ϕ and postcondition ψ, and $\{\phi\}C\{\psi\}$ is valid when the program C is partially correct with respect to the precondition ϕ and postcondition ψ. In what follows this chapter, we wil use the partial correctness terminology to describe the inference systems.

2.3.3 *Hoare calculus*

The theoretical tool that allow one to reason about Hoare triples and, therefore, reason about programs and their specifications, is the Hoare calculus.

The Hoare calculus is an inference system for reasoning about Hoare triples. Its rules are given in Figure 3. This system describes, in a formal way, the use of preconditions, postconditions, and loop invariants in order to verify programs, and is usually described as an alternative semantics of the underlying programming language.

We provide a detailed explanation about each rule of the system.

- $skip$ - we recall that $skip$ is the *do-nothing* command. Thus, it preserves the truth of assertions. Without any remaining command, the remaining prove goal is the implication between the precondition and the postcondition.
- *assign* - this rule states that a postcondition ψ can be ensured for an assignment $x := e$ by replacing every occurence of x by e in it.
- *seq* - analysing the *seq* rule, there is an intermediate assertion θ that is, simultaneously, precondition for the last command and postcondition to the first command, working as the *connection*

$$\text{(skip)}\frac{\phi \Rightarrow \psi}{\{\phi\}skip\{\psi\}}$$

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

$$\text{(seq)}\frac{\{\phi\}C_1\{\theta\} \quad \{\theta\}C_2\{\psi\}}{\{\phi\}C_1; C_2\{\psi\}}$$

$$\text{(while)}\frac{\{\theta \wedge b\}C\{\theta\}}{\{\theta\}while\ b\ do\{\theta\}C\{\theta \wedge \neg b\}}$$

$$\text{(if)}\frac{\{\phi \wedge b\}C_t\{\psi\} \quad \{\phi \wedge \neg b\}C_f\{\psi\}}{\{\phi\}if\ b\ then\ C_t\ else\ C_f\{\psi\}}$$

$$\text{(conseq)}\frac{\{\phi\}C\{\psi\}}{\{\phi'\}C\{\psi'\}} \quad if\ \phi' \Rightarrow \phi\ and\ \psi \Rightarrow \psi'$$

Figure 3: Inference system of Hoare logic

between the two commands. Consequently, to deal with the axiom one needs to prove that there exists an assertion that can be used as postcondition for C_1 (with precondition ϕ) and also as precondition for C_2 (with postcondition ψ).

- *while* - the aim of this rule is to prove that θ is a loop invariant for the given while-loop. Informally, one needs to prove that θ holds at the beginning of the loop, during its execution and after it.

- if - the description of the rule comes along with the execution of the conditional statement $if\ b\ then\ C_t\ else\ C_f$: if the condition b holds, then C_t is executed and the precondition can be combined as $\phi \wedge b$. Conversely, if b does not hold, then C_f is executed and the precondition can be combined as $\phi \wedge \neg b$. The postcondition ψ needs to hold independently of the execution flow.

- *conseq* - the *conseq* rule is not actually a *program* rule since it does not deal with any instruction by itself. However, it can be extremely useful because it allows one to strengthen the precondition or to weaken the postcondition in order to deal easily with some program or instruction. Therefore, the *conseq* rule introduces some ambiguity to the Hoare calculus, since it can always be applied to any Hoare triple to obtain another Hoare triple.

It is extremely important that the inference system of Figure 3 cannot be used to derive invalid Hoare triples. It is obvious to see that, if one could do so, the system would be completely useless and it would not serve the purposes of software verification.

The crucial step, when dealing with formal software verification and with Hoare logic, is to come up with a well annotated program. One must find the specifications that correctly suit the program and the purposes of the program.

2.4 PROBABILISTIC HOARE LOGIC

We have shown an inference system to deal with deterministic programs. However, when dealing with cryptographic software, we want to be able to reason about probabilistic programs, since cryptographic software contains probabilistic states. For example, every key generation algorithm has some randomness attached to it and random samplings are an important step of most of the cryptographic primitives. Therefore, we need to extend the *While* language (Section 2.2) and the Hoare logic (Figure 2.3) in such way that enables reasoning about probabilistic states. In what follows this section, we will follow the work of J. I. den Hartog in den Hartog and de Vink (2002).

2.4.1 *A probabilistic While language - pWhile*

A deterministic state is a function that maps a variable to a value. A probabilistic state gives the probability of being in a given deterministic state. Thus, a probabilistic state can be seen as a (countable) weighted set of deterministic states $\rho_1 \cdot \sigma_1 + \rho_2 \cdot \sigma_2 + ... + \rho_n \cdot \sigma_n$, for some probabilities $\rho_1, \rho_2, ..., \rho_n$ and some deterministic states $\sigma_1, \sigma_2, ..., \sigma_n$. The probability of being in the deterministic state σ_i is $\rho_i \in [0, 1]$. The sum of all probabilities is always greater than zero and at most one. A probability ρ less than one indicates that this execution point may not be always reached.

Consequently, we can also define a probabilistic state as being a probabilistic distribution, that maps every possible deterministic state to its probability of happening. Therefore, the occurence of deterministic steps in a program can be defined as samplings from some probabilistic distribution.

To deal with this additional feature, the *While* language described in Section 2.2 is, then, naturally extended as follows, resulting in a new language - *pWhile*:

$$\begin{array}{lll} \text{Comm} \ni C & :: & ... \\ & | & x^\tau := \$d^\tau \\ & | & ... \end{array}$$

Figure 4: Extensions to the *While* language - samplings

It is obvious to note that the Hoare logic is not suitable to reason about programs written in *pWhile*. Recalling the Hoare calculus of Figure 3, there is no program rule that deals with probabilistic samplings. We introduce in the following sections an extension to the original Hoare logic, called probabilistic Hoare logic, and a new inference system that is extended with the new sampling command and with probabilistic reasoning.

2.4.2 *Bounded Hoare triples*

Similarly to what happens in Hoare logic, to be able to formally reason about the correctness of a probabilistic program, one uses Hoare triples. The only difference is that these are now bounded to some probability.

$$[\{\phi\} C \{\psi\}] \leq \delta$$

The above present bounded Hoare triple means that, if ϕ holds at a given state and C is executed in that state, then ψ will hold at the resulting state with probability at most δ.

2.4.3 *Probabilistic Hoare calculus*

To reason about deterministic states and programs, we have shown a derivation system in Figure 3 - the Hoare calculus. In this section, we introduce the probabilist Hoare calculus, that contemplates the use of probabilities, as well as introduces a new program rule to deal with probabilistic samplings. This inference system is presented in Figure 5.

$$\text{(skip)}\frac{\phi \Rightarrow \psi \quad \delta = 1}{[\{\phi\} skip \{\psi\}] \leq \delta}$$

$$\text{(assign)}\frac{\phi \Rightarrow \psi[e/x] \quad \delta = 1}{[\{\phi\} x := e \{\psi\}] \leq \delta}$$

$$\text{(seq)}\frac{\begin{array}{c}[\{\phi\}C_1\{\theta\}] \leq \delta_1 \quad [\{\theta\}C_2\{\psi\}] \leq \delta_2 \\ [\{\phi\}C_1\{\neg\theta\}] \leq \delta_3 \quad [\{\neg\theta\}C_2\{\psi\}] \leq \delta_4 \\ \delta_1 \cdot \delta_2 + \delta_3 \cdot \delta_4 \leq \delta\end{array}}{[\{\phi\}C_1; C_2\{\psi\}] \leq \delta}$$

$$\text{(while)}\frac{\begin{array}{c}[\{\phi\}C'\{\theta \wedge \forall M.(\theta \wedge 0 \leq e \Rightarrow \neg b) \wedge (\theta \wedge \neg b \Rightarrow \phi)\}] \leq \delta \\ \forall k.[\{\theta \wedge b \wedge e = k\}C\{\theta \wedge e < k\}] = 1\end{array}}{[\{\phi\}C';\ while\ b\ do \{\theta\}C\{\psi\}] \leq \delta}$$

$$\text{(if)}\frac{[\{\phi \wedge b\}C_t; C\{\psi\}] \leq \delta \quad [\{\phi \wedge \neg b\}C_f; C\{\psi\}] \leq \delta}{[\{\phi\} if\ b\ then\ C_t\ else\ C_f;\ C\{\psi\}] \leq \delta}$$

$$\text{(sample)}\frac{\begin{array}{c}\delta_1 \cdot \delta_2 + \delta_3 \cdot \delta_4 \leq \delta \\ [\{\phi\}C\{\theta\}] \leq \delta_1 \\ \theta \Rightarrow \mu\, d\, v \leq \delta_2 \wedge (\forall v, v \in supp\ d \Rightarrow \psi[v/x] \Rightarrow p\, v) \\ [\{\phi\}C\{\neg\theta\}] \leq \delta_3 \\ \neg\theta \Rightarrow \mu\, d\, p \leq \delta_4 \wedge (\forall v, v \in supp\ d \Rightarrow \psi[v/x] \Rightarrow p\, v)\end{array}}{[\{\phi\}C;\ x := \$d\{\psi\}] \leq \delta}$$

$$\text{(conseq)}\frac{[\{\phi\}C\{\psi\}] \leq \delta}{[\{\phi'\}C\{\psi'\}] \leq \delta} \quad if\ \phi' \Rightarrow \phi\ and\ \psi \Rightarrow \psi'$$

Figure 5: Inference system of probabilistic Hoare logic

We provide a detailed explanation about each rule of the system.

- $skip$ - we state that this rule is similar to the one of the Hoare logic: being the *do-nothing* command, one is left to prove that the precondition implies the postcondition with probability $\delta = 1$ - meaning that the implication is always true.

- $assign$ - the explanation of this rule follows the same paradigm as the previous one: one needs to prove that the precondition continues to imply the postcondition, when all the occurrences of x are replaced by e in the postcondition.

- seq - the seq probabilistic rule contemplates all the possible scenarios in the execution of a probabilistic program, with the middle assertion θ also presented in the seq deterministic rule. From top to bottom and left to right, the first bounded Hoare triple corresponds to the execution of C_1 that produces θ with probability at most δ_1, the second one corresponds to the execution of C_2 (now with θ as a precondition) that produces ψ with probability at most δ_2 and remaining two are similar to the previous ones, but when θ does not hold. Finally, the sum of the probabilities of each execution ($\delta_1 \cdot \delta_2$ standing for the execution of C_1 and C_2 that results in a state where ψ is valid such that at the end of the execution of C_1 the resulting state models θ and $\delta_3 \cdot \delta_4$ standing for the execution of C_1 and C_2 that results in a state where ψ is valid such that at the end of the execution of C_1 the resulting state does not model θ) must be a valid probability, i.e., must be at most δ.

- $while$ - in this formula, M stands for the set of variables that may be modified by C, θ is the loop invariant and e is the variant expression, used to prove termination. This rule is also similar to the $while$ rule for the original Hoare calculus in Figure 3, with the slightly fact that one adds variant reasoning to it: the expressions $0 \leq e$, $e = k$ and $e < k$ are used with that objective, combined with all the variables from the set M.

- if - the if rules are similar in both systems (Figure 3 and Figure 5), except that the Hoare triples are now bounded.

- $sample$ - in this rule, θ is the intermediate assertion between C and $x := \$d$, being $\$d$ the operation of sampling a value from the distribution d. The predicate $supp$ checks if some value v is in the support of a distribution d - v is defined in the distribution - and $\mu\, d\, p$ is the probability of sampling some value v from d such that $p\, v$ holds, for some predicate p. Therefore, the $sample$ rule can be seen as mix of the seq and $assign$ rule: it combines the probability of the value extracted from d to follow the predicate p, replaces all the occurrences of the variable being assigned in the postcondition and then continues the reasoning to the following command. The same principle applies when the value sampled from d does not follows the predicate p.

Similar rules hold by substituting every occurrence of the comparison operator $\leq$ by the operators $=$, $<$, $>$ or $\geq$.

2.5 PROBABILISTIC RELATIONAL HOARE LOGIC

In cryptography, security proofs are usually structured using sequences of games Shoup (2004). Informally, the principle behind this concept is that one is able to builds a reduction proof by proving indistinguishability between games. One starts by defining the original game, that corresponds to the desired cryptographic property, and then build a sequence of games that differ from the previous ones in some indistinguishability step. At the end of the sequence, one is able to reduce the security break of the scheme to some assumption.

2.5.1 *Relational Hoare logic*

Relational Hoare logic Benton (2004) is a variant of the original Hoare logic that reasons about two programs. Instead of assertions that denote predicates on states and judgements which say that terminating execution of a command in a state satisfying a precondition will yield a state satisfying a postcondition, we are now dealing with a pair of commands, that if a precondition holds before the execution of both commands in different programs, then the postcondition will hold at the end of the execution of both commands. The assertions in relational Hoare logic need to specify to which of the commands some variable belongs.

The calculus for this logic follows directly from the one of the original Hoare logic, showed in Section 2.3, and can be consulted in Figure 6. Informally, one can take the two commands to whom some rule is being applied and separately apply an Hoare logic rule to the two commands.

We introduce the notation $C_1 \sim C_2$ to denote the comparison between two programs and a new style for Hoare triples $\{\phi\}C_1 \sim C_2\{\psi\}$ that denote that if a precondition holds before the execution of both commands in different programs, then the postcondition will hold at the end of the execution of these commands. We will denote program C_1 as *left program*, with memory $\{1\}$ - and program C_2 as *right program*, with memory $\{2\}$.

We presented the inference system for the relational Hoare logic by showing *one side* rules - when the rule is only applied to just one program, independently of the instruction of the other program - and *both side* rules - when the same instruction appears in both programs.

Relational Hoare logic is mostly used when proving equivalence between programs. For example, one could define a program and prove it correct with respect to some definition. After, one could iterate over it in order to produce a more efficient program. Therefore, in order to prove that the efficient program behaves as the original program, one would perform an equivalence proof between the two programs and use relational Hoare logic to discard that proof. In cryptography, this kind of interpretation is used to prove equivalences between cryptographic games, so that one is able to reduce the security of some cryptographic primitive to some mathematical assumption.

$$\text{(skip)}\frac{\phi \Rightarrow \psi}{\{\phi\} skip \sim skip \{\psi\}}$$

$$\text{(seq)}\frac{\{\phi\} C_1 \sim C_1' \{\theta\} \quad \{\theta\} C_2 \sim C_2' \{\psi\}}{\{\phi\} C_1; C_2 \sim C_1'; C_2' \{\psi\}}$$

$$\text{(assign - one side)}\frac{\phi \Rightarrow \psi[x/e]}{\{\phi\} x := e \sim skip \{\psi\}}$$

$$\text{(assign - both sides)}\frac{\phi \Rightarrow \psi[x, x'/e, e']}{\{\phi\} x := e \sim x' := e' \{\psi\}}$$

$$\text{(if - one side)}\frac{\{\phi \wedge b\} C_t \sim C \{\psi\} \quad \{\phi \wedge \neg b\} C_f \sim C \{\psi\}}{\{\phi\} if\ b\ then\ C_t\ else\ C_f \sim C \{\psi\}}$$

$$\text{(if - both sides)}\frac{\begin{array}{c}\phi \Rightarrow b \Leftrightarrow b' \\ \{\phi \wedge b \wedge b'\} C_t \sim C_t' \{\psi\} \quad \{\phi \wedge \neg b \wedge \neg b'\} C_f \sim C_f' \{\psi\}\end{array}}{\{\phi\} if\ b\ then\ C_t\ else\ C_f \sim if\ b'\ then\ C_t'\ else\ C_f' \{\psi\}}$$

$$\text{(while - one side)}\frac{\{\theta \wedge b\} C \sim C' \{\theta\} \quad \forall k, \{\theta \wedge b \wedge e = k\} C \{\theta \wedge e < k\}}{\{\theta\} while\ b\ do \{\theta\} C \sim C' \{\neg b \wedge \theta\}}$$

$$\text{(while - both sides)}\frac{\phi \equiv b \Leftrightarrow b' \wedge \theta \quad \{b \wedge b' \wedge \theta\} C \sim C' \{\theta\}}{\{\theta\} while\ b\ do \{\theta\} C \sim while\ b'\ do \{\theta\} C' \{\neg b \wedge \neg b' \wedge \theta\}}$$

$$\text{(conseq)}\frac{\{\phi\} C \sim C' \{\psi\}}{\{\phi'\} C \sim C' \{\psi'\}} \quad if\ \phi' \Rightarrow \phi\ and\ \psi \Rightarrow \psi'$$

Figure 6: Relation Hoare logic calculus

2.5.2 *Probabilistic relational Hoare calculus*

We showed, in the previous section, a relational Hoare logic to reason about relations between two deterministic programs. However, we have seen in Section 2.4 that deterministic programs do not encompass the necessary means to deal with cryptographic programs, since these are probabilistic programs. Consequently, the relational Hoare logic can not be used in order to reason about relations between two probabilistic programs.

Similar to what was done for the probabilistic Hoare logic (Section 2.4), we extend the relational Hoare logic with probabilistic reasoning, ending up with a probabilistic relational Hoare logic.

The only difference between this calculus and the previously presented in Figure 6 is the addition of the sampling rule. We stick to the presentation of that rule in Figure 7. In contrast to what was presented for the Hoare logic (Section 2.3) and probabilistic Hoare logic (Section 2.4), the probabilistic relational Hoare logic does not introduce *bounds* to relational Hoare logic. The predicate *lossless* attests if all the probabilities of a given distribution sum up to 1.

$$\text{(sampling - one side)}\frac{\phi = lossless\ d \quad \forall v \in supp\ d, Q[x_1/v]}{\{\phi\} x = \$d \sim skip \{\psi\}}$$

$$\text{(sampling - both sides)}\frac{\phi = \forall v \in supp\ d. \psi[x_1, x_2'/v, f\ v]}{\{\phi\} x := \$d \sim x' := \$d' \{\psi\}}$$

Figure 7: Sampling rules for probabilistic relational Hoare logic

2.5.3 *Provable security*

Provable security Goldwasser and Micali (1984) is a paridgm that aims to provide a new methodology to verify rigorously the security of cryptographic systems. A provable security argument is developed in three steps:

1. Define the security goal - i.e., what is the security that one aims to achieve - and an adversary model - i.e., what are the adversary capabilities.
2. Define the cryptographic system - i.e., *implement* the cryptographic scheme - and the security assumptions in which the primitive relies on.
3. Prove that any attack to the cryptographic system can be used to efficiently break the security assumption. This proof is done by reduction.

A direct consequence of provable security is *practice-oriented provable security* Bellare and Rogaway (1993). The central goal of practice-oriented provable security is the analysis of efficient cryp-

tographic systems that can be used for practical purposes and to prove its security in what respects some security parameter and assumption.

An important realisation of practice-oriented provable security is the *code-based approach* Bellare and Rogaway (2004). This paradigm makes use of programming languages techniques to organise proofs in a systematic way. The security hypotheses and security goals are defined in terms of the probability of some event with respect to some implementation and specification of a probabilistic program, called *games*. These programs are written in *pWhile*.

Formally, a cryptographic game is a probabilistic program that takes as input an initial memory , i.e. a mapping from variables to values, and returns a sub-distribution on memories. A sub-distribution on a memory is a map $d : \mathcal{M} \mapsto [0,1]$, such that $\sum_{m \in \mathcal{M}} d\ m \leq q$, between elements of the memory and some probability.

In the provable security setting, proofs are made by reduction. A reduction argument develops in the following way. Assume that there is a cryptographic system that is secure under some mathematical assumption. Let $\mathcal{A}$ be an adversary against the security of the system. The goal of the reduction proof is to show that there exists an adversary $\mathcal{B}$ such that the success probability of $\mathcal{A}$ in the attack of the system is upper bounded by a function of the success probability of $\mathcal{B}$ in breaking the security of the assumption. The adversary $\mathcal{B}$ invokes $\mathcal{A}$ as a subprocedure and both of them are also defined as probabilistic programs.

In the code-based approach, proofs are structured as sequences of games, so that transitions between games are simple and easy to justify. There are two kinds of transitions:

- Transitions based on indistinguishability - a small change is made such that, if detected by the adversary, would imply an efficient method of distinguishing between two distributions that are indistinguishable (either statistically or computationally).
- Transitions based on failure events - two games i and $i+1$ proceed identically unless a certain "failure event" occurs.

A transition between two games G_1 and G_2 establishes an inequality of the form

$$Pr[G_1, m_1 : A] \leq Pr[G_2, m_2 : B] + \epsilon$$

where $Pr[G, m : E]$ denotes the probability of the occurence of the event E in the execution of game G with initial memory m and ϵ is a negligible arithmetic expression. The proof concludes by combining the inequalities proven for each transition to bound the success probability of the reduction.

2.5.4 *Verifiable security*

Verified security Bellare and Rogaway (2004); Halevi (2005) is a new approach to perform security proofs of cryptographic systems. It follows the same principles as practice-oriented provable security

but revisits its realisation from a formal verification perspective. In the verified security approach, proofs are built and verified using verification tools, such as CertiCrypt Barthe et al. (2009) or EasyCrypt.

2.6 SOFTWARE FORMAL VERIFICATION

Software formal verification is the act of proving the correctness of a given program with respect to a certain formal specification. There are a lot of techniques that provide means to formally verify properties over programs. For example, model checking techniques decompose a program in all its possible states and prove the desired property for all states of the program. Verification can also be labeled as dynamic or static. In the first case, verification is performed at run-time, i.e., properties about programs are verified when the program is executing. As for the second case, verification is carried out before the code is executed and has obvious performance and reliability advantages.

In this work, we follow an approach that uses deductive reasoning based on Hoare logic, called *deductive software verification*. In deductive verification, properties of programs are specified by terms of *contracts* - pre- and postconditions - that are annotated into programs. Therefore, the main purpose of software deductive verification is to, given a precondition ϕ, a postcondition ψ and a program P such that ϕ and ψ are contracts of P, build a derivation tree using the inference systems previously shown to check if P has the desired property accordingly to its contract.

Briefly, the architecture of a deductive verification infrastructure consists of a verification condition generator (VCGen) and a proof tool, which may be either an automatic theorem prover (SMT solver) or an interactive proof assistant (like, for example, COQ The Coq development team (2004)). The workflow is as follows. The VCGen starts by reading the annotated code and producing a set of verification conditions that are generated according to the inference system being used. After, these verification conditions are sent to the proof tool to be discharged. If all the verification conditions (or proof obligations) are proved correct, then the annotated program is correct with respect to its specification.

2.6.1 *Safety properties*

Correctness properties are of extreme importance for Computer Science: they assure that some program behaves in accordance to some desired behaviour. However, one can specify a program, annotate it and prove it correct in respect to some specification but, if the program was to be executed, it would not finish its execution because it will end in some *error* statement due to some *bug* in the code. For example, assume the following CAO program P

```
def safety_error () : vector[10] of int {
  def a : vector[10] of int;
```

```
  seq i := 0 to 11 {
    a[i] := 1;
  }

  return a;
}
```

which declares a function that changes the elements of an integer array, of size 10, to the value 1. Suppose that this function is annotated with the precondition $\phi = true$ and postcondition $\psi = \forall x, 0 \leq x < 10 \Rightarrow a[i] = 10$. The Hoare triple $\{\phi\}P\{\psi\}$ would be valid and one was able to prove it. Nevertheless, the execution of the program would result in an *out of bounds* error - the program would try to access memory that does not belong to the memory allocated by the array. This is the problem that safety conditions aim to solve.

The safety properties of programs can be captured by *safety-sensitive* Hoare triples. In Figure 8, we extend the original Hoare logic to deal with the safety of expressions, using the predicate $safe$, defined in Figure 9. A safe expression e is one that does not result in some *error* state. For example, e is safe if it does not contain invalid accesses to elements of an array or invalid operations (like divisions by zero).

Besides being presented only for the Hoare logic, the safety-sensitive Hoare calculus can easily be applied in the context of probabilistic Hoare logic or probabilistic relational Hoare logic. For example, a safety-sensitive $if - bothsides$ rule for the probabillistc Hoare logic could be rewritten as follows

$$\text{(if - both sides)}\frac{\begin{array}{c}\phi \Rightarrow b \Leftrightarrow b' \\ \{\phi \wedge b \wedge b'\}C_t \sim C_t'\{\psi\} \quad \{\phi \wedge \neg b \wedge \neg b'\}C_f \sim C_f'\{\psi\}\end{array}}{\{\phi\}if\ b\ then\ C_t\ else\ C_f \sim if\ b'\ then\ C_t'\ else\ C_f'\{\psi\}} if\ \phi \Rightarrow safe(b) \wedge \phi \Rightarrow safe(b')$$

Finally, we extend the safety predicate to reason about probabilistic distributions and provide a safety-sensitive rule for probabilistic samplings. We state that a probabilistic distribution is safe if the weight of the distribution is 1.

$$safe(d) = lossless\ d$$

A safety-sensitive sampling rule can be defined as follows

$$\text{(sample)}\frac{\begin{array}{c}\delta_1 \cdot \delta_2 + \delta_3 \cdot \delta_4 \leq \delta \\ [\{\phi\}C\{\theta\}] \leq \delta_1 \\ \theta \Rightarrow \mu\, d\, v \leq \delta_2 \wedge (\forall v, v \in supp\ d \Rightarrow \psi[v/x] \Rightarrow p\, v) \\ [\{\phi\}C\{\neg\theta\}] \leq \delta_3 \\ \neg\theta \Rightarrow \mu\, d\, p \leq \delta_4 \wedge (\forall v, v \in supp\ d \Rightarrow \psi[v/x] \Rightarrow p\, v)\end{array}}{[\{\phi\}C;\ x := \$d\{\psi\}] \leq \delta} if\ \phi \Rightarrow safe(d)$$

$$\text{(skip)}\frac{\phi \Rightarrow \psi}{\{\phi\} skip \{\psi\}}$$

$$\text{(assign)}\frac{\phi \Rightarrow \psi[e/x]}{\{\phi\} x := e \{\psi\}} if\ \phi \Rightarrow safe(e)$$

$$\text{(seq)}\frac{\{\phi\} C_1 \{\theta\} \quad \{\theta\} C_2 \{\psi\}}{\{\phi\} C_1; C_2 \{\psi\}}$$

$$\text{(while)}\frac{\{\theta \wedge b \wedge safe(b)\} C \{\theta \wedge safe(b)\}}{\{\phi\} while\ b\ do \{\theta\} C \{\psi\}} if\ \phi \Rightarrow (\theta \wedge safe(b)) \wedge (\theta \wedge safe(b) \wedge \neg b \Rightarrow \psi)$$

$$\text{(if)}\frac{\{\phi \wedge b\} C_t \{\psi\} \quad \{\phi \wedge \neg b\} C_f \{\psi\}}{\{\phi\} if\ b\ then\ C_t\ else\ C_f \{\psi\}} if\ \phi \Rightarrow safe(b)$$

$$\text{(conseq)}\frac{\{\phi\} C \{\psi\}}{\{\phi'\} C \{\psi'\}} \quad if\ \phi' \Rightarrow \phi\ and\ \psi \Rightarrow \psi'$$

Figure 8: Safety-sensitive Hoare calculus

$$\begin{aligned}
\mathbf{safe}(true) &= true \\
\mathbf{safe}(false) &= true \\
\mathbf{safe}(b) &= true \\
\mathbf{safe}(e) &= true \\
\mathbf{safe}(e_1 \circ e_2) &= \mathbf{safe}(e_1) \wedge \mathbf{safe}(e_2),\ where \circ \in \{+, \times, -, **, <, >, \leq, \geq, =, != \} \\
\mathbf{safe}(e_1 \dagger e_2) &= \mathbf{safe}(e_1) \wedge \mathbf{safe}(e_2) \wedge e_2 \neq 0,\ where \dagger \in \{/, mod\}
\end{aligned}$$

Figure 9: Safe predicate

2.6.2 *Extensions to Hoare logic for realistic programs*

In the previous sections we have presented, a simple imperative language - *While* language - and an extension to it, that includes probabilistic operations in it - *pWhile* language. Both languages support simple commands, that allow the description of somewhat complex programs. However, they lack the support of some operations and commands in order to be used in more realistic contexts.

2.6.2.1 *Arrays*

The inclusion of the *array* type is very important in every programming language: they provide a container type, that encompasses one or more values indexed by integers, and the access operation over arrays has time complexity $O(1)$.

We extend the languages with the a new expression (an array element) and a new command (array modification).

$$
\begin{array}{lcl}
\text{Exp}_\tau \ni e^\tau & :: & \ldots \\
 & | & a^\tau[x^{int}] \\
 & | & \ldots \\
\text{Comm} \ni C & :: & = \ldots \\
 & | & a^\tau[e_1^{int}] = e_2^\tau \\
 & | & \ldots
\end{array}
$$

Figure 10: Extensions to the *While* language - arrays

The Hoare logic, probabilistic Hoare logic and probabilistic relational Hoare logic rules for array assignment can be found in Figure 11, Figure 12 and Figure 13, respectively. In these rules, given an array a, of type τ *array*, an integer expression e_1 and a τ expression e_2, the operation $a[e_1 \leftarrow e_2]$ returns a new array in which the value indexed by e_1 has been replaced by e_2.

$$\text{(array assign)}\frac{\phi \Rightarrow \psi[a[e_1 \leftarrow e_2]/a]}{\{\phi\}a[e_1] := e_2\{\psi\}}$$

Figure 11: Array assigment rule for Hoare logic

$$\text{(array assign)}\frac{\phi \Rightarrow \psi[a[e_1 \leftarrow e_2]/a] \wedge \delta = 1}{[\{\phi\}a[e_1] := e_2\{\psi\}] \leq \delta}$$

Figure 12: Array assigment rule for probabilistic Hoare logic

We also extend the safety-sensitive Hoare calculus to provide a definition for what is a safety array operation. Naturally, one will check if the access of the array does not result in some out of bounds error and if the index expression is also safe. The rule can be consulted in Figure 14.

$$\text{(array assign - one side)}\frac{\phi \Rightarrow \psi[a[e_1 \leftarrow e_2]/a]}{\{\phi\}a[e_1] := e_2 \sim skip\{\psi\}}$$

$$\text{(array assign - both side)}\frac{\phi \Rightarrow \psi[a[e_1 \leftarrow e_2], a'[e_1' \leftarrow e_2']/a, a']}{\{\phi\}a[e_1] := e_2 \sim a'[e_1'] := e_2'\{\psi\}}$$

Figure 13: Array assigment rule for probabilistic relational Hoare logic

$$\text{(array assign)}\frac{\phi \Rightarrow \psi[a[e_1 \leftarrow e_2]/a]}{\{\phi\}a[e_1] := e_2\{\psi\}} if\ \phi \Rightarrow safe(a[e_1]) \wedge \phi \Rightarrow safe(e_2)$$

Figure 14: Safety-sensity array assignement rule

The safe predicate is also extended in the following way.

$$\begin{aligned} \mathbf{safe}(a) &= true \\ \mathbf{safe}(a[e]) &= \mathbf{safe}(a) \wedge \mathbf{safe}(e) \wedge 0 \leq e < \mathbf{length}a \end{aligned}$$

where a is an array variable and $len(a)$ is the operation that returns the size of the array a.

2.6.2.2 *Procedure calls*

A procedure is a sequence of program instructions that performs a specific task, packaged as a unit. Procedures are very important in the context of Computer Science because they allow the development of modular code, since one can program different procedures, with different objectives and then invoke then as many times as needed. Particularly, in cryptography, procedures are important because they allow the complete and formal description of cryptographic primitives, as well as the invokation of adversaries and oracles.

However, procedures are very challenging from a verification point of view. The challenges include, for example, the treatment of recursive calls, the reasoning about parameters of the functions and the inclusion of multiple function calls on expressions. In the context of this project, we are only interested in procedures that have their own specifications and that are not recursive. Additionally, we do not allow the definition of function calls as expressions.

Informally, what one needs to ensure is that the parameters that are being used to call the procedure match its precondition, prove the validity of the Hoare triple composed of the body of the function and its annotations and then procede with the analysis of the original function. The rule that denotes this behaviour is showed in Figure 15, where $\overrightarrow{y}$ are the concrete parameters with which the function f is being invoked, $\overrightarrow{p}$ are the formal parameters of f, res_f is the result variable of f and $\overrightarrow{m}$ is the efect of f, i.e., the set of variables modified by f.

$$\text{(proc call)}\ \frac{\{\phi\}C\{\phi_f[\overrightarrow{y}\,/\,\overrightarrow{p}] \wedge \forall v, \forall \overrightarrow{z}, \psi_f[v, \overrightarrow{z}\,/res_f, \overrightarrow{m}] \Rightarrow \psi[v, \overrightarrow{z}\,/x, \overrightarrow{m}]\} \quad \{\phi_f\}f\{\psi_f\}}{\{\phi\}C; x := f(\overrightarrow{y})\{\psi\}}$$

Figure 15: Procedure call rule for Hoare logic

With respect to probabilistic Hoare logic, the rule is similar, with the addition of the bound to the function being called.

$$\text{(proc call)}\ \frac{\{\phi\}C\{\phi_f[\overrightarrow{y}\,/\,\overrightarrow{p}] \wedge \forall v, \forall \overrightarrow{z}, \psi_f[v, \overrightarrow{z}\,/res_f, \overrightarrow{m}] \Rightarrow \psi[v, \overrightarrow{z}\,/x, \overrightarrow{m}]\} \quad [\{\phi_f\}f\{\psi_f\}] \leq \delta}{[\{\phi\}C; x := f(\overrightarrow{y})\{\psi\}] \leq \delta}$$

Figure 16: Procedure call rule for probabilistic Hoare logic

Finally, we present the rule for probabilistic relation Hoare logic in Figure 17.

$$\text{proc call}\ \frac{\{\phi_f\}f \sim f'\{\psi_f\} \quad \phi \Rightarrow \phi_f[\overrightarrow{p_f}, \overrightarrow{p'_f}\,/\,\overrightarrow{y}, \overrightarrow{y'}] \quad \forall v\ v', \forall \overrightarrow{z}\ \overrightarrow{z'}, \psi_f[v, v', \overrightarrow{z}, \overrightarrow{z'}\,/res_f, res_{f'}, \overrightarrow{m}, \overrightarrow{m'}] \Rightarrow \psi[v, v', \overrightarrow{z}, \overrightarrow{z'}\,/x, x', \overrightarrow{m}, \overrightarrow{m'}]}{\{\phi\}x := f(\overrightarrow{y}) \sim x' := f'(\overrightarrow{y'})\{\psi\}}$$

Figure 17: Procedure call rule for probabilistic relational Hoare logic

2.6.3 *Focus on automation vs focus on interactivity*

In what comes to program verification, there are a lot of ways to deal with it. One can define a recursive procedure wp, like defined in [Dijkstra (1997)], develop a VCGen using it and then rely on SMT solvers to discard all the generated proof obligations. Note that this method would imply the removal of the *conseq* tactic, since it introduces ambiguity on the system. In constrast to this approach, one can define an interactive VCGen, in which the user specifies which tactic is to be used next, until he ends up with an empty program, which can then be transformed into a first-order logic implication, that can after be proved using first-order logic reasoning or even be sent to some SMT solver.

In the context of this project, we analyse two different approaches to the verification of cryptographic software, by having the CAOVerif tool [Almeida et al. (2014)] to operate with two different backends: one that relies on the Frama-C platform and one that relies on EasyCrypt.

The first approach is already developed and can be consulted in [Almeida et al. (2014)]. An annotated CAO program is translated into C code, annotated accordingly to ACSL [Baudin et al. (2010)]. After, the annotated C code is parsed by Frama-C, with the Jessie plug-in, to generate all the proof obligations (including safety properties) that could be discharged using the Why tool, by relying on one or more SMT solver or by using the Coq proof assistant. Besides having some clear advantages (like automation), this approach could be painful when dealing with bigger programs, as the amount of proof obligations could turn the tool impractical.

The second approach - developed in the context of this project - consists in mapping an annotated CAO code into an EasyCrypt script and then perform all the necessary proofs there. EasyCrypt has an internal VCGen that relies on the probabilistic relation Hoare logic, that is used to build proof trees and discharge proof obligations interactivly and/or using SMT solvers. Using this approach, the problem of the explosion of proof obligations would be eliminated, the CAOVerif tool would be relying of a platform specific to the domain of cryptography, but the degree of automation would be lower.

2.7 STATE OF THE ART TOOLS FOR VERIFICATION OF CRYPTOGRAPHIC SOFTWARE

Cryptographic domain specific languages gained a lot of interest recently. Nowadays, for cryptographic software, there is the need to program at a high level fixture (more suitable to cryptography) and be able to reason about high level implementations, as well as to generate code from those implementations. In the context of this work, the domain specific language being used in CAO, but one can find more cryptographic domain specific languages, like the Cryptol language Erkök and Matthews (2009), developed by Galois.

There are some important differences between CAO and Cryptol. Cryptol is a functional language, which contrasts with the imperative style of CAO. This can be seen as an advantage for the CAO language, since most of the cryptographic primitives are described in an imperative style and contemplate some features that are hard to replicate in functional languages. Additionally, Cryptol incorporates a proof system that allows one to perform proofs on the specified algorithms. However, this proof system is not ideal, since it makes use of model checking techniques, which leads to several inefficiency problems. The CAOVerif tool - suject of this project - allows one to annotate CAO programs and perform proofs about CAO programs with respect to those annotations. This tool makes use of the Jessie plug-in for Frama-C and still has some overheads that this project aims to solve. Both languages allow code generation: Cryptol contemplates Haskell and C code extraction, while CAO is able to generate only C code.

As cryptographic proofs became essentially unverifiable, there was a migration to computer-aided proofs by cryptographers. Therefore, some tools were developed to assist the construction and verification of cryptographic proofs. EasyCrypt Barthe et al. (2011b) is an SMT-based verification framework for building and verifying security proofs of cryptographic constructions, following a

language-based approach. Its main application is the construction and verification of game-based cryptographic proofs. In order to do so, EasyCrypt admits probabilistic computations with adversarial code, and makes use of the probabilistic relational Hoare logic (Section 2.5.2) to reason about those computations.

ZooCrypt Barthe et al. (2013) is a tool for automatically analyzing and synthesizing secure instances within a well-defined class of cryptographic constructions, such as padding-based encryption schemes (public-key encryption schemes that are built from one-way trapdoor permutations and random oracles) or public-key encryption schemes based on bilinear pairings.

CryptoVerif Blanchet (2005) is an automatic protocol prover sound in the computational model. It can prove secrecy and correspondences (e.g. authentication). The generated proofs are by sequences of games. Essentially, CryptoVerif allows one to perform more ”high-level” proofs: for example, one is not able to prove that a given encryption scheme is IND-CPA secure but can prove that if that encryption scheme is IND-CPA secure and if it is combined with a MAC scheme and is UF-CMA secure, both originate an IND-CCA encryption scheme.

Recently, a new approach was made in order to perform cryptographic proofs. F^* Swamy et al. (2011) is a new ML-like functional programming language designed with program verification in mind. It has a powerful refinement type-checker that discharges verification conditions using the Z3 SMT solver. F^* has been successfully used to verify cryptographic protocol implementations. Again, one is not able to perform security proofs like in EasyCrypt, however, one can use F^* to automatically prove important security properties of implementations.

3

CAO SPECIFICATION

CAO Barbosa et al. (2012); (editor) is a programming language developed towards the automatic production of highly efficient target code, subjected to security-aware optimisations. Therefore, it does not encompass some high-level features of other imperative languages but has features specific to the development of cryptographic software like its type system (that contemplates, for example, the *ring* $\mathbb{Z}_n$ type) as well as a close syntax to the one used in the cryptographic standards.

In what follows this chapter, we will follow the formal CAO specification that can found in Barbosa et al. (2012) and in (editor).

3.1 CAO SYNTAX

The formalisation of the CAO syntax is presented in Figure 18.

Most the CAO binary operators are the same as their C equivalents. However, note that CAO contemplates some operators that are widely used in cryptography: multiplicative exponentiation for integers, residue class groups and fields ($**$), bit-wise operators such as conjunction (&), inclusive-disjunction (|) and exclusive-disjunction (^), shift ($<<$ and $>>$) and shift-rotate ($< \mid$ and $\mid >$) operators for bit strings and vectors, concatenation operation (@) for bit strings and vectors and the boolean logic exclusive-disjunction (^^). Additionally, note also that the CAO syntax is very similar to the C language.

3.2 CAO TYPE SYSTEM

The CAO type system was developed with two main focuses. On one hand, to be able to check whether a program was written according to the syntactic rules of Figure 18 and that it does not violate any type checking rule, thus ensuring that no invalid program would be evaluated. On the other hand, the type checker collects important information about expressions in a CAO program, that can then be used by other features of the CAO tool chain, like the C code extraction.

CAO is a dependently typed language Paulin-Mohring (2014). This means that the CAO type checker is aware of type parameters in the data types of the language. Concretely, the CAO type

$$
\begin{array}{rcl}
e & :: & L \mid x \mid -e \mid e_1{+}e_2 \mid e_1{-}e_2 \mid e_1{*}e_2 \mid e_1 \ / \ e_2 \mid e_1 \ \% \ e_2 \mid e_1{**} \ e_2 \\
& \mid & e.fi \mid (t) \ e \mid fp(e_1, \ldots, e_n) \\
& \mid & e_1 == e_2 \mid e_1 \ != \ e_2 \mid e_1 < e_2 \mid e_1 > e_2 \mid e_1 <= e_2 \mid e_1 >= e_2 \mid e_1 \| e_2 \mid e_1 \ \&\& \ e_2 \mid ! \ e \mid e_1 \hat{} \hat{} \ e_2 \\
& \mid & e_1[e_2] \mid e_1[e_2..e_3] \mid e_1[e_2, e_3] \mid e_1[e_2, e_3..e_4] \mid e_1[e_2..e_3, e_4] \mid e_1[e_2..e_3, e_4..e_5] \\
& \mid & \sim e \mid e_1 \ \& \ e_2 \mid e_1 \hat{} \ e_2 \mid e_1 | e_2 \mid e_1 \ll e_2 \mid e_1 \gg e_2 \mid e_1 < | e_2 \mid e_1 | > e_2 \mid e_1 \ @ \ e_2 \\
l & :: & x \mid l[e] \mid l[e_1..e_2] \\
& \mid & l[e_1, e_2] \mid l[e_1, e_2..e_3] \mid l[e_1..e_2, e_3] \mid l[e_1..e_2, e_3..e_4] \\
& \mid & l.fi \\
c & :: & dv \\
& \mid & l_1, \ldots, l_n := e_1, \ldots, e_m; \\
& \mid & \textbf{return} \ e_1, \ldots, e_n; \\
& \mid & fp(e_1, \ldots, e_n); \\
& \mid & \textbf{if} \ (e) \ \{ \ c_1; \ldots; c_n \ \} \\
& \mid & \textbf{if} \ (e) \ \{ \ c_{11}; \ldots; c_{1n} \ \} \ \textbf{else} \ \{ \ c_{21}; \ldots; c_{2m} \ \} \\
& \mid & \textbf{while} \ (e) \ \{ \ c_1; \ldots; c_n \ \} \\
& \mid & \textbf{seq} \ x := e_1 \ \textbf{to} \ e_2 \ \textbf{by} \ e_3 \ \{ \ c_1; \ldots; c_n \ \} \\
& \mid & \textbf{seq} \ x := e_1 \ \textbf{to} \ e_2 \ \{ \ c_1; \ldots; c_n \ \} \\
dt & :: & \textbf{typedef} \ tid \ := \ t; \\
& \mid & \textbf{typedef} \ sid \ := \ \textbf{struct} \ [\ \textbf{def} \ fi_1 : t_1; \ldots; \textbf{def} \ fi_n : t_n; \ ]; \\
dv & :: & \textbf{def} \ x : t; \\
& \mid & \textbf{def} \ x_1, \ldots, x_n : t; \\
& \mid & \textbf{def} \ x : t := e; \\
& \mid & \textbf{def} \ x : t := \{ \ e_1, \ldots, e_n \ \}; \\
dfp & :: & \textbf{def} \ fp \ (x_1 : t_1, \ldots, x_n : t_n) : rt \ \{ \ c_1; \ldots; c_m \ \} \\
rt & :: & \textbf{void} \mid t_1, \ldots, t_n \\
t & :: & \textbf{int} \mid \textbf{bool} \\
& \mid & \textbf{unsigned bits} \ [e] \mid \textbf{signed bits} \ [e] \\
& \mid & \textbf{mod} \ [e] \mid \textbf{mod} \ [t/pol] \\
& \mid & \textbf{vector} \ [e] \ \textbf{of} \ t \mid \textbf{matrix} \ [e_1, e_2] \ \textbf{of} \ t \\
& \mid & tid \mid sid \\
pg & :: & dv \mid dt \mid dfp \mid pg_1 \ pg_2
\end{array}
$$

Figure 18: CAO formal syntax

system explicitly includes as type parameters the sizes of containers such as vectors, matrices and bit strings. Consequently, typing of complex operations over these containers, including concatenation and extensional assignment, statically checks the compatibility of these parameters. Dependent types are also present in the mathematical types contemplated by CAO. The type system stores information about the values that define the moduli types (integer or polynomial) so that it is possible to validate the consistency of complex mathematical expressions. This important feature of CAO makes it possible to detect several common run-time errors, like invalid accesses to some vector. However, the language looses some flexibility, since the type parameters (sizes of container types and integers and polynomials that define rings, fields or extension fields) need to be fully determined, implying that type declarations in CAO can depend on arithmetic expressions using constants stored in the environment Δ, described next.

In this section, we will provide a detailed explanation of the CAO type system. We will follow the same syntax as the one presented in Section 2.1.

PRELIMINARIES In its typechecking mechanism, CAO makes use of two environments:

- Γ - environment that associates all the identifiers (like variables' names or functions' names) to their types. This environment is divided into two environments Γ_G and Γ_L that store information about global and local identifiers, respectively. This distinction is important in order to keep track of the scope and visibility of identifiers when typing.
- Δ - environment that collects all integer constants and that associates them with their value.

Notation $\Gamma[x :: \tau]$ is used to extend the environment Γ with a new variable x of type τ, providing that x is not in the original environment (i.e., $x \notin \Gamma$). Similarly, $\Delta[x := n]$ is used to extend the environment Δ with a new constant x with value n, also provided that x is not in the domain of environment Δ. Notation $\Gamma(x)$ and $\Delta(x)$ represent, respectively, the type and the integer value associated with identifier x, assuming that x belongs to the domain of the respective environment.

Finally, symbol $\bullet$ represents a possible return type. The objective of this symbol is to distinguish the cases when a block has explicitly executed a return statement (in which case we use the type of the parameter passed to the return statement) from the cases where no return statement has been executed in the the block (in which case we use the $\bullet$ to signal this situation).

DATA TYPES We start by first presenting the CAO types (Table 1) that are used in the type checking rules. Note that these types are different from the ones used in type declarations of CAO programs, as the laters are mapped into the ones of Table 1, that refer to the formalisation of the CAO types. For example, when an int type appears in CAO syntax, it will be mapped into an Int type. We denote by $\mathcal{A}$ the set of algebraic types - types for which addition, multiplication and symmetric operators are closed.

Void	The empty type
Int	Arbitrary precision integers
Bool	Booleans
UBits[i]	Unsigned bit strings of length i
SBits[i]	Signed bit strings of length i
Mod[n]	Ring or field defined by n
Mod[τ / pol]	Extension field defined by τ / pol
Vector[i] of τ	Vector of size i with elements of type τ
Matrix[i, j] of α	Matrix of size $i \times j$ with elements of type α
$\mathcal{A} = \{\text{Int}, \text{Mod}[n], \text{Mod}[\tau \ / \ pol], \text{Matrix}[i, j] \text{ of } \alpha, \alpha \in \mathcal{A}\}$	

Table 1: CAO types formalisation

Syntactic types are translated into formal types accordingly to a judgment of the form $\Delta \vdash_t t \rightsquigarrow \tau$, where t is a syntactic type and τ is a formal type. Note that this judgement only depends on environment Δ, since the type checker needs to be able to evaluate all the integer expressions in the type declarations. Thus, the type checker makes use of a partial function φ_Δ, that computes the value of an integer expression e in the context of Δ. Naturally, if e is not able to be determined, then the type checking will fail because it will not be able to correctly define a type. The function is defined in Figure 19.

$$\begin{aligned} \phi_\Delta(n) &= n & \phi_\Delta(x) &= \Delta(x),\ x \in dom\ \Delta \\ \phi_\Delta(-e) &= -\phi_\Delta(e) & \phi_\Delta(e_1 \dagger e_2) &= \phi_\Delta(e_1) \dagger \phi_\Delta(e_2) \\ \phi_\Delta(e_1 ** e_2) &= (\phi_\Delta(e_1))^{(\phi_\Delta(e_2))} & \phi_\Delta(e_1 \,\%\, e_2) &= \phi_\Delta(e_1) \mod \phi_\Delta(e_2) \end{aligned}$$

$$\text{for } \dagger \in \{+, -, *, /\}.$$

Figure 19: Definition of function φ_Δ

The definiton of the type translation can be found in Figure 20.

Note that algebraic types are the only ones that can be used to construct matrices, since algebraic operations are defined in the matrix type.

FUNCTION CLASSIFICATION CAO type checker classifies functions with respect to their interaction with global variables:

- *Pure* - do not depend on global variables in any way, i.e., no global variable value is read or modified. In order to be considered *pure*, a function must only call other *pure* functions.
- *Read-only* - can read values from global variables but can not modify them. Again, a *read-only* function can only call other *read-only* or *pure* functions.
- *Procedures* - can read and assign values from/to global variables.

$$\frac{}{\Delta \vdash_t \mathsf{int} \rightsquigarrow \mathsf{Int}} \qquad \frac{}{\Delta \vdash_t \mathsf{bool} \rightsquigarrow \mathsf{Bool}}$$

$$\frac{\phi_\Delta(e) = n}{\Delta \vdash_t \mathsf{unsigned\ bits}\ [e] \rightsquigarrow \mathsf{UBits}[n]}\ n \geq 1 \qquad \frac{\phi_\Delta(e) = n}{\Delta \vdash_t \mathsf{signed\ bits}\ [e] \rightsquigarrow \mathsf{SBits}[n]}\ n \geq 1$$

$$\frac{\phi_\Delta(e) = n}{\Delta \vdash_t \mathsf{mod}\ [e] \rightsquigarrow \mathsf{Mod}[n]}\ n \geq 2 \qquad \frac{\Delta \vdash_t t \rightsquigarrow \tau}{\Delta \vdash_t \mathsf{mod}\ [t/pol] \rightsquigarrow \mathsf{Mod}[\tau/pol]}$$

$$\frac{\phi_\Delta(e) = n \quad \Delta \vdash_t t \rightsquigarrow \tau}{\Gamma, \Delta \vdash_t \mathsf{vector}\ [e]\ \mathsf{of}\ t \rightsquigarrow \mathsf{Vector}\ [n]\ \mathsf{of}\ \tau}\ n \geq 1$$

$$\frac{\phi_\Delta(e_1) = n \quad \phi_\Delta(e_2) = m \quad \Delta \vdash_t t \rightsquigarrow \alpha}{\Delta \vdash_t \mathsf{matrix}\ [e_1, e_2]\ \mathsf{of}\ t \rightsquigarrow \mathsf{Matrix}\ [n, m]\ \mathsf{of}\ \alpha}\ \alpha \in \mathcal{A}, n \geq 1, m \geq 1$$

Figure 20: Type translation

$\mathbf{t_1}$	$\mathbf{t_2}$	**Condition**
$\mathsf{UBits}[n]$	Int	
$\mathsf{SBits}[n]$	Int	
τ	$\mathsf{Mod}[\tau'/pol]$	$\vdash_\leq \tau \leq \tau'$
$\mathsf{Vector}[n]$ of τ_1	$\mathsf{Vector}[n]$ of τ_2	$\vdash_\leq \tau_1 \leq \tau_2$
$\mathsf{Matrix}\ [i,j]$ of α_1	$\mathsf{Matrix}\ [i,j]$ of α_2	$\vdash_\leq \alpha_1 \leq \alpha_2$ and $\alpha_1, \alpha_2 \in \mathcal{A}$

Table 2: Type coercion relation, $\vdash_\leq t_1 \leq t_2$

This classification is also applied when type checking expressions.

TYPE COERCIONS Type coercions are implicit type conversions, that allow a programmer to use terms of some type when other type is expected, providing that the type of the term being used can be converted into the expected type. An example of a type coercion in CAO can be found when dealing with bit strings of a given size, that can be coerced to the integer type, and thus one can use the integer operators when dealing with bit strings. Essentially, one type τ_1 can be coerced into another type τ_2 if the set of all values of τ_1 can be seen as a subset of the set of all values of τ_2.

To deal with coercions, the CAO type system defines a reflexive coercion relation $\leq$ and a new judgement $\vdash_\leq$. This relation is summarised in Table 2.

Additionally, it may be the case when two terms with different types are being used in some expression but are coercible to a common type. To capture the situation, CAO defines a new operator on types $\uparrow$ that finds the least upper bound of the types to which its arguments are coercible.

$$\tau_1 \uparrow \tau_2 = min\{\tau | \tau_1 \leq \tau \wedge \tau_2 \leq \tau\}$$

Having this operation requires the $\leq$ relation to be reflexive, transitive and anti-symmetric.

$\mathbf{t_1}$	$\mathbf{t_2}$	**Condition**
Int	Bits $[i]$	
Int	Mod $[n]$	
Vector $[i]$ of τ_1	Mod $[\tau_2/pol]$	$\vdash_c \tau_1 \Rightarrow \tau_2$ and $i = degree(pol)$
Mod $[\tau_1/pol]$	Vector $[i]$ of τ_2	$\vdash_c \tau_1 \Rightarrow \tau_2$ and $i = degree(pol)$
Matrix $[1, j]$ of α	Vector $[j]$ of τ	$\vdash_c \alpha \Rightarrow \tau$ and $\alpha \in \mathcal{A}$
Vector $[i]$ of τ	Matrix $[i, 1]$ of α	$\vdash_c \tau \Rightarrow \alpha$ and $\alpha \in \mathcal{A}$
Vector $[i]$ of τ_1	Vector $[i]$ of τ_2	$\vdash_c \tau_1 \Rightarrow \tau_2$
Matrix $[i, j]$ of α_1	Matrix $[i, j]$ of α_2	$\vdash_c \alpha_1 \Rightarrow \alpha_2$ and $\alpha_1, \alpha_2 \in \mathcal{A}$

Table 3: A few cases for the cast relation, $\vdash_c t_1 \Rightarrow t_2$.

CASTS A cast is a mechanism that allows a programmar to explicitly convert values from one type into another. Casts are similar to coercions, except that now it is the programmer that indicates to which type he/she wants some value to be converted, instead of leaving this job to the type checker.

However, not all casts are possible: the set of admissible type cast operations has been carefully designed to account for those conversions that are conceptually meaningful in the mathematical sense and/or are important for the implementation of cryptographic software in a natural way.

Similar to what was done for coercions, the CAO type checker contemplates a cast relation $\Rightarrow$ - Table 3 - and the associated judgement $\vdash_{\Rightarrow}$. The typing rule for the cast relation is the following

$$\frac{\vdash_{\leq} \tau_1 \leq \tau_2}{\vdash_c \tau_1 \Rightarrow \tau_2} \qquad \frac{\Delta \vdash_t t \rightsquigarrow \tau \quad \Gamma, \Delta \vdash e :: (\tau', \mathsf{c}) \quad \vdash_c \tau' \Rightarrow \tau}{\Gamma, \Delta \vdash (t)\ e :: (\tau, \mathsf{c})}$$

LITERALS In CAO, one can specify integer, ring, boolean and bit string literals. The typechecking rules for these literals are very simple and can be found in Figure 21.

$$\frac{}{\Gamma \vdash \mathsf{true} :: \mathsf{Bool}} \qquad \frac{}{\Gamma \vdash \mathsf{false} :: \mathsf{Bool}} \qquad \frac{}{\Gamma \vdash \mathsf{b}(0|1)^i :: \mathsf{Bits}[i]}$$

$$\frac{}{\Gamma \vdash (0..9)^* :: \mathsf{Int}} \qquad \frac{}{\Gamma \vdash [(0..9)^*] :: \mathsf{Mod}[n]}$$

Figure 21: Typechecking rules for literals

VARIABLES, FUNCTION CALLS AND STRUCT PROJECTIONS In expressions, only side-effects free functions (*pure* and *read-only*) can be used. When typechecking a struct projection, there is the need to check the expression that defines the struct and the field that is being projected (see the type checking rules in Figure 22).

$$\frac{\Gamma_G(x) = \tau}{\Gamma_G, \Gamma_L, \Delta \vdash x :: (\tau, \mathsf{ReadOnly})} \quad x \in dom(\Gamma_G)$$

$$\frac{\Gamma_L(x) = \tau}{\Gamma_G, \Gamma_L, \Delta \vdash x :: (\tau, \mathsf{Pure})} \quad x \in dom(\Gamma_L)$$

$$\frac{\Gamma_G(f) = ((\tau_1, \ldots, \tau_n) \to \tau, \mathsf{c}) \quad \Gamma_G, \Gamma_L, \Delta \vdash e_1 \leq (\tau_1, \mathsf{c_1}) \quad \ldots \quad \Gamma_G, \Gamma_L, \Delta \vdash e_n \leq (\tau_n, \mathsf{c_n})}{\Gamma_G, \Gamma_L, \Delta \vdash f(e_1, \ldots, e_n) :: (\tau, \max(\mathsf{c}, \mathsf{c_1}, \ldots, \mathsf{c_n}))} \quad \mathsf{c} < \mathsf{Procedure}, f \in dom(\Gamma_G)$$

$$\frac{\Gamma_G(fi) = (\tau_1 \to \tau_2, \mathsf{Pure}) \quad \Gamma_G, \Gamma_L, \Delta \vdash e :: (\tau_1, \mathsf{c})}{\Gamma_G, \Gamma_L, \Delta \vdash e.fi :: (\tau_2, \mathsf{c})} \quad fi \in dom(\Gamma_G)$$

Figure 22: Typechecking rules for variables, function calls and struct projections

BOOLEAN OPERATIONS Operations over booleans are define by the typechecking rules of Figure 23.

$$\frac{\Gamma, \Delta \vdash e_1 :: (\tau_1, \mathsf{c_1}) \quad \Gamma, \Delta \vdash e_2 :: (\tau_2, \mathsf{c_2}) \quad \tau_1 \uparrow \tau_2 = \tau}{\Gamma, \Delta \vdash e_1 \oplus e_2 :: (\mathsf{Bool}, \max(\mathsf{c_1}, \mathsf{c_2}))} \quad \oplus \in \{==, != \}$$

$$\frac{\Gamma, \Delta \vdash e_1 \leq (\mathsf{Int}, \mathsf{c_1}) \quad \Gamma, \Delta \vdash e_2 \leq (\mathsf{Int}, \mathsf{c_2})}{\Gamma, \Delta \vdash e_1 \oplus e_2 :: (\mathsf{Bool}, \max(\mathsf{c_1}, \mathsf{c_2}))} \quad \oplus \in \{<, \leq, >, \geq\}$$

$$\frac{\Gamma, \Delta \vdash e_1 \leq (\mathsf{Bool}, \mathsf{c_1}) \quad \Gamma, \Delta \vdash e_2 \leq (\mathsf{Bool}, \mathsf{c_2})}{\Gamma, \Delta \vdash e_1 \oplus e_2 :: (\mathsf{Bool}, \max(\mathsf{c_1}, \mathsf{c_2}))} \quad \oplus \in \{||, \&\&, \hat{}\}$$

$$\frac{\Gamma, \Delta \vdash e \leq (\mathsf{Bool}, \mathsf{c})}{\Gamma, \Delta \vdash !e :: (\mathsf{Bool}, \mathsf{c})}$$

Figure 23: Typechecking rules for boolean operations

ARITHMETIC OPERATIONS Arithmetic operations are defined to the set of algebraic type defined in Table 1. However, there are some subtleties that need to be taken into account for some specific types. For example, the division operation for modular types is not defined if the integer that parametrises the structure is not prime and thus does not construct a field. The program type checking rules for arithmetic operations can be found in Figure 24.

BIT STRING OPERATIONS All the bit string operation are closed over the same representation, i.e., one can not mix signed and unsigned bit strings unless through an explicit cast. The bit-wise operations can only be used with bit strings of the same size. The concatenation, selection and range

$$\frac{\Gamma,\Delta \vdash e_1 :: (\alpha_1, \mathsf{c}_1) \quad \Gamma,\Delta \vdash e_2 :: (\alpha_2, \mathsf{c}_2) \quad \alpha_1 \uparrow \alpha_2 = \alpha}{\Gamma,\Delta \vdash e_1 \oplus e_2 :: (\alpha, \max(\mathsf{c}_1, \mathsf{c}_2))} \quad \alpha, \alpha_1, \alpha_2 \in \mathcal{A} \quad \oplus \in \{+, -\}$$

$$\frac{\Gamma,\Delta \vdash e_1 :: (\alpha_1, \mathsf{c}_1) \quad \Gamma,\Delta \vdash e_2 :: (\alpha_2, \mathsf{c}_2) \quad \alpha_1 \uparrow \alpha_2 = \alpha}{\Gamma,\Delta \vdash e_1 * e_2 :: (\alpha, \max(\mathsf{c}_1, \mathsf{c}_2))} \quad \alpha, \alpha_1, \alpha_2 \in \mathcal{A}$$

$$\frac{\Gamma,\Delta \vdash e_1 :: (\mathsf{Matrix}[i,j] \text{ of } \alpha_1, \mathsf{c}_1) \quad \Gamma,\Delta \vdash e_2 :: (\mathsf{Matrix}[j,k] \text{ of } \alpha_2, \mathsf{c}_2) \quad \alpha_1 \uparrow \alpha_2 = \alpha}{\Gamma,\Delta \vdash e_1 * e_2 :: (\mathsf{Matrix}[i,k] \text{ of } \alpha, \max(\mathsf{c}_1, \mathsf{c}_2))}$$

where $\alpha, \alpha_1, \alpha_2 \in \mathcal{A}$

$$\frac{\Gamma,\Delta \vdash e_1 :: (\tau_1, \mathsf{c}_1) \quad \Gamma,\Delta \vdash e_2 :: (\tau_2, \mathsf{c}_2) \quad \vdash_{\leq} \tau_1 \leq \mathsf{Int} \quad \vdash_{\leq} \tau_2 \leq \mathsf{Int}}{\Gamma,\Delta \vdash e_1 \oplus e_2 :: (\mathsf{Int}, \max(\mathsf{c}_1, \mathsf{c}_2))} \quad \oplus \in \{+, -\}$$

$$\frac{\Gamma,\Delta \vdash e_1 :: (\tau_1, \mathsf{c}_1) \quad \Gamma,\Delta \vdash e_2 :: (\tau_2, \mathsf{c}_2) \quad \vdash_{\leq} \tau_1 \leq \mathsf{Int} \quad \vdash_{\leq} \tau_2 \leq \mathsf{Int}}{\Gamma,\Delta \vdash e_1 * e_2 :: (\mathsf{Int}, \max(\mathsf{c}_1, \mathsf{c}_2))}$$

$$\frac{\Gamma,\Delta \vdash e_1 :: (\alpha, \mathsf{c}_1) \quad \Gamma,\Delta \vdash e_2 \leq (\mathsf{Int}, \mathsf{c}_2)}{\Gamma,\Delta \vdash e_1 ** e_2 :: (\alpha, \max(\mathsf{c}_1, \mathsf{c}_2))} \quad \alpha \in \mathcal{A}$$

$$\frac{\Gamma,\Delta \vdash e_1 :: (\tau, \mathsf{c}_1) \quad \vdash_{\leq} \tau \leq \mathsf{Int} \quad \Gamma,\Delta \vdash e_2 \leq (\mathsf{Int}, \mathsf{c}_2)}{\Gamma,\Delta \vdash e_1 ** e_2 :: (\mathsf{Int}, \max(\mathsf{c}_1, \mathsf{c}_2))} \quad \tau \notin \mathcal{A}$$

$$\frac{\Gamma,\Delta \vdash e_1 :: (\mathsf{Matrix}[i,i] \text{ of } \alpha, \mathsf{c}_1) \quad \Gamma,\Delta \vdash e_2 \leq (\mathsf{Int}, \mathsf{c}_2)}{\Gamma,\Delta \vdash e_1 ** e_2 :: (\mathsf{Matrix}[i,i] \text{ of } \alpha, \max(\mathsf{c}_1, \mathsf{c}_2))} \quad \alpha \in \mathcal{A}$$

$$\frac{\Gamma,\Delta \vdash e_1 \leq (\mathsf{Int}, \mathsf{c}_1) \quad \Gamma,\Delta \vdash e_2 \leq (\mathsf{Int}, \mathsf{c}_2)}{\Gamma,\Delta \vdash e_1 \,/\, e_2 :: (\mathsf{Int}, \max(\mathsf{c}_1, \mathsf{c}_2))}$$

$$\frac{\begin{array}{c}\Gamma,\Delta \vdash e_1 :: (\mathsf{Mod}\ [m_1], \mathsf{c}_1) \quad \Gamma,\Delta \vdash e_2 :: (\mathsf{Mod}\ [m_2], \mathsf{c}_2) \\ \mathsf{Mod}\ [m_1] \uparrow \mathsf{Mod}\ [m_2] = \mathsf{Mod}\ [m]\end{array}}{\Gamma,\Delta \vdash e_1 \,/\, e_2 :: (\mathsf{Mod}\ [m], \max(\mathsf{c}_1, \mathsf{c}_2))}$$

where m_1, m_2 can be of the form n or t/pol

$$\frac{\Gamma,\Delta \vdash e :: (\alpha, \mathsf{c})}{\Gamma,\Delta \vdash -e :: (\alpha, \mathsf{c})} \quad \alpha \in \mathcal{A}$$

$$\frac{\Gamma,\Delta \vdash e_1 \leq (\mathsf{Int}, \mathsf{c}_1) \quad \Gamma,\Delta \vdash e_2 \leq (\mathsf{Int}, \mathsf{c}_2)}{\Gamma,\Delta \vdash e_1 \,\%\, e_2 :: (\mathsf{Int}, \max(\mathsf{c}_1, \mathsf{c}_2))}$$

Figure 24: Typechecking rules for arithmetic operations

selection operators work in the natural way by construction of a bit string with the appropriate size. Figure 25 describes the rules for type checking bit string operations.

$$\frac{\Gamma,\Delta \vdash e_1 :: (\mathsf{Bits}[i],\mathsf{c}_1) \quad \Gamma,\Delta \vdash e_2 :: (\mathsf{Bits}[i],\mathsf{c}_2)}{\Gamma,\Delta \vdash e_1 \oplus e_2 :: (\mathsf{Bits}[i],\max(\mathsf{c}_1,\mathsf{c}_2))} \quad \oplus \in \{|,\&,\hat{}\}$$

$$\frac{\Gamma,\Delta \vdash e :: (\mathsf{Bits}[i],\mathsf{c})}{\Gamma,\Delta \vdash \sim e :: (\mathsf{Bits}[i],\mathsf{c})}$$

$$\frac{\Gamma,\Delta \vdash e_1 :: (\mathsf{Bits}[i],\mathsf{c}_1) \quad \Gamma,\Delta \vdash e_2 \leq (\mathsf{Int},\mathsf{c}_2)}{\Gamma \vdash e_1 \oplus e_2 :: (\mathsf{Bits}[i],\max(\mathsf{c}_1,\mathsf{c}_2))} \quad \oplus \in \{\ll,\gg,<|,|>\}$$

$$\frac{\Gamma,\Delta \vdash e_1 :: (\mathsf{Bits}[i],\mathsf{c}_1) \quad \Gamma,\Delta \vdash e_2 :: (\mathsf{Bits}[j],\mathsf{c}_2)}{\Gamma,\Delta \vdash e_1 @ e_2 :: (\mathsf{Bits}[i+j],\max(\mathsf{c}_1,\mathsf{c}_2))}$$

$$\frac{\Gamma,\Delta \vdash e_1 :: (\mathsf{Bits}[i],\mathsf{c}_1) \quad \Gamma,\Delta \vdash e_2 \leq (\mathsf{Int},\mathsf{c}_2)}{\Gamma,\Delta \vdash e_1[e_2] :: (\mathsf{Bits}[1],\max(\mathsf{c}_1,\mathsf{c}_2))}$$

$$\frac{\Gamma,\Delta \vdash e :: (\mathsf{Bits}[k],\mathsf{c}) \quad \phi_\Delta(e_1) = i \quad \phi_\Delta(e_2) = j}{\Gamma,\Delta \vdash e[e_1..e_2] :: (\mathsf{Bits}[j-i+1],\mathsf{c})} \quad k > j, j \geq i \geq 0$$

Figure 25: Typechecking rules for bit string operations

VECTOR OPERATIONS Vectors are the generic container type and they contemplate a series of operations. As for bit strings, one can perform shifts, concatenations, selection and range selection. These operations are defined similar to how they were defined for bit strings. The type checking rules to be applied in vector operations are presented in Figure 26.

$$\frac{\Gamma,\Delta \vdash e_1 :: (\mathsf{Vector}[i] \text{ of } \tau,\mathsf{c}_1) \quad \Gamma,\Delta \vdash e_2 \leq (\mathsf{Int},\mathsf{c}_2)}{\Gamma,\Delta \vdash e_1 \oplus e_2 :: (\mathsf{Vector}[i] \text{ of } \tau,\max(\mathsf{c}_1,\mathsf{c}_2))} \quad \oplus \in \{\ll,\gg,<|,|>\}$$

$$\frac{\Gamma,\Delta \vdash e_1 :: (\mathsf{Vector}[i] \text{ of } \tau_1,\mathsf{c}_1) \quad \Gamma,\Delta \vdash e_2 :: (\mathsf{Vector}[j] \text{ of } \tau_2,\mathsf{c}_2) \quad \tau_1 \uparrow \tau_2 = \tau}{\Gamma,\Delta \vdash e_1 @ e_2 :: (\mathsf{Vector}[i+j] \text{ of } \tau,\max(\mathsf{c}_1,\mathsf{c}_2))}$$

$$\frac{\Gamma,\Delta \vdash e_1 :: (\mathsf{Vector}[i] \text{ of } \tau,\mathsf{c}_1) \quad \Gamma,\Delta \vdash e_2 \leq (\mathsf{Int},\mathsf{c}_2)}{\Gamma,\Delta \vdash e_1[e_2] :: (\tau,\max(\mathsf{c}_1,\mathsf{c}_2))}$$

$$\frac{\Gamma,\Delta \vdash e :: (\mathsf{Vector}[k] \text{ of } \tau,\mathsf{c}) \quad \phi_\Delta(e_1) = i \quad \phi_\Delta(e_2) = j}{\Gamma,\Delta \vdash e[e_1..e_2] :: (\mathsf{Vector}[j-i+1] \text{ of } \tau,\mathsf{c})} \quad k > j, j \geq i \geq 0$$

Figure 26: Typechecking rules for vector operations

MATRIX OPERATIONS We have seen the description of arithmetic operations over matrices in Figure 24. Yet, it is missing the formulation of the typechecking rules for matrix access and range selection, formalised in Figure 27.

$$\frac{\Gamma,\Delta \vdash e_1 :: (\mathsf{Matrix}[i,j] \text{ of } \alpha, \mathsf{c}_1) \quad \Gamma,\Delta \vdash e_2 \leq (\mathsf{Int}, \mathsf{c}_2) \quad \Gamma,\Delta \vdash e_3 \leq (\mathsf{Int}, \mathsf{c}_3)}{\Gamma,\Delta \vdash e_1[e_2,e_3] :: (\alpha, \max(\mathsf{c}_1,\mathsf{c}_2,\mathsf{c}_3))}$$

where $\alpha \in \mathcal{A}$

$$\frac{\Gamma,\Delta \vdash e :: (\mathsf{Matrix}[u,v] \text{ of } \alpha, \mathsf{c}) \quad \phi_\Delta(e_1) = i \quad \phi_\Delta(e_2) = j \quad \phi_\Delta(e_3) = k \quad \phi_\Delta(e_4) = n}{\Gamma,\Delta \vdash e[e_1..e_2, e_3..e_4] :: (\mathsf{Matrix}[j-i+1, n-k+1] \text{ of } \alpha, \mathsf{c})}$$

where $u > j, j \geq i \geq 0, v > n, n \geq k \geq 0, \alpha \in \mathcal{A}$

$$\frac{\Gamma,\Delta \vdash e :: (\mathsf{Matrix}[u,v] \text{ of } \alpha, \mathsf{c}) \quad \Gamma,\Delta \vdash e_1 \leq (\mathsf{Int}, \mathsf{c}_1) \quad \phi_\Delta(e_2) = k \quad \phi_\Delta(e_3) = n}{\Gamma,\Delta \vdash e[e_1, e_2..e_3] :: (\mathsf{Matrix}[1, n-k+1] \text{ of } \alpha, \max(\mathsf{c}, \mathsf{c}_1))}$$

where $v > n, n \geq k \geq 0, \alpha \in \mathcal{A}$

$$\frac{\Gamma,\Delta \vdash e :: (\mathsf{Matrix}[u,v] \text{ of } \alpha, \mathsf{c}) \quad \phi_\Delta(e_1) = i \quad \phi_\Delta(e_2) = j \quad \Gamma,\Delta \vdash e_3 \leq (\mathsf{Int}, \mathsf{c}_3)}{\Gamma,\Delta \vdash e[e_1..e_2, e_3] :: (\mathsf{Matrix}[j-i+1, 1] \text{ of } \alpha, \max(\mathsf{c}, \mathsf{c}_3))}$$

where $u > j, j \geq i \geq 0, \alpha \in \mathcal{A}$

Figure 27: Typechecking rules for matrix operations

STATEMENTS There is one important aspect that needs to be considered when typechecking statements of the CAO language: some statements may modify the environments. For example, the declaration of a constant retrieves not only a typed statement but also a new type environment (extended with the new constant and its associated type) and a new constant environment (extended with the new constant and its associated value). To account this issue, we introduce a new operator $\Vdash$ that denotes type judgments of statements that may change the environment relations. We include statement rules in Figures 28 and 29.

PROGRAMS As shown in Figure 18, a CAO program consists of procedure, function, variable, constant and struct declarations. When analysing a program, the type checker follows a *non-lazy* approach, meaning that it will proceed immediately to type check the contents of the declaration. The type checking rules for declarations can be found in Figure 30

$$\frac{\Delta \vdash_t t \rightsquigarrow \tau}{\Gamma, \Delta \models_\rho \mathsf{def}\ x : t :: (\bullet, \mathsf{Pure}, \Gamma[x :: \tau])} \quad x \notin dom(\Gamma)$$

$$\frac{\Delta \vdash_t t \rightsquigarrow \tau}{\Gamma, \Delta \models_\rho \mathsf{def}\ x_1, \ldots, x_n : t :: (\bullet, \mathsf{Pure}, \Gamma[x_1 :: \tau, \ldots, x_n :: \tau])}$$

where $x_1, \ldots, x_n \notin dom(\Gamma), x_i \neq x_j$ for $1 \leq i \leq n$ and $1 \leq j \leq n$

$$\frac{\Delta \vdash_t t \rightsquigarrow \tau \quad \Gamma, \Delta \vdash e \leq (\tau, \mathsf{cc})}{\Gamma, \Delta \models_\rho \mathsf{def}\ x : t := e :: (\bullet, \mathsf{cc}, \Gamma[x :: \tau])} \quad x \notin dom(\Gamma)$$

$$\frac{\Delta \vdash_t t \rightsquigarrow \mathsf{Vector}\ [n]\ \mathsf{of}\ \tau \quad \Gamma, \Delta \vdash e_1 \leq (\tau, \mathsf{cc}_1) \ldots \Gamma, \Delta \vdash e_n \leq (\tau, \mathsf{cc}_n)}{\Gamma, \Delta \models_\rho \mathsf{def}\ x : t := \{e_1, \ldots, e_n\} :: (\bullet, \max(\mathsf{cc}_1, \ldots, \mathsf{cc}_n), \Gamma[x :: \mathsf{Vector}\ [n]\ \mathsf{of}\ \tau])}$$

where $x \notin dom(\Gamma)$

$$\frac{\Delta \vdash_t t \rightsquigarrow \mathsf{Matrix}\ [i, j]\ \mathsf{of}\ \alpha \quad \Gamma, \Delta \vdash e_1 \leq (\alpha, \mathsf{cc}_1) \ldots \Gamma, \Delta \vdash e_n \leq (\alpha, \mathsf{cc}_n)}{\Gamma, \Delta \models_\alpha \mathsf{def}\ x : t := \{e_1, \ldots, e_n\} :: (\bullet, \max(\mathsf{cc}_1, \ldots, \mathsf{cc}_n), \Gamma[x :: \mathsf{Matrix}\ [i, j]\ \mathsf{of}\ \alpha])}$$

where $\alpha \in \mathcal{A}, x \notin dom(\Gamma), i \times j = n$

$$\frac{\begin{array}{c}\Gamma, \Delta \vdash l_1 :: (\tau_1, \mathsf{cl}_1) \quad \ldots \quad \Gamma, \Delta \vdash l_n :: (\tau_n, \mathsf{cl}_n) \\ \Gamma, \Delta \vdash e_1 \leq (\tau_1, \mathsf{c}_1) \quad \ldots \quad \Gamma, \Delta \vdash e_n \leq (\tau_n, \mathsf{c}_n)\end{array}}{\Gamma, \Delta \models_\tau l_1, \ldots, l_n := e_1, \ldots, e_n :: (\bullet, \max(\mathsf{cl}_1 \ldots, \mathsf{cl}_n, \mathsf{c}_1, \ldots, \mathsf{c}_n), \Gamma)}$$

$$\frac{\Gamma, \Delta \vdash l_1 :: (\tau_1, \mathsf{cl}_1) \quad \ldots \quad \Gamma, \Delta \vdash l_n :: (\tau_n, \mathsf{cl}_n) \quad \Gamma, \Delta \vdash e \leq ((\tau_1, \ldots, \tau_n), \mathsf{c})}{\Gamma, \Delta \models_\tau l_1, \ldots, l_n := e :: (\bullet, \max(\mathsf{cl}_1, \ldots, \mathsf{cl}_n, \mathsf{c}), \Gamma)}$$

$$\frac{\begin{array}{c}\Gamma_G(fp) = ((\tau_1, \ldots, \tau_n) \rightarrow (\tau_{n+1}, \ldots, \tau_m), \mathsf{Procedure}) \\ \Gamma_G, \Gamma_L, \Delta \vdash l_{n+1} :: (\tau_{n+1}, \mathsf{cl}_1) \quad \ldots \quad \Gamma_G, \Gamma_L, \Delta \vdash l_m :: (\tau_m, \mathsf{cl}_m) \\ \Gamma_G, \Gamma_L, \Delta \vdash e_1 \leq (\tau_1, \mathsf{c}_1) \quad \ldots \quad \Gamma_G, \Gamma_L, \Delta \vdash e_n \leq (\tau_n, \mathsf{c}_n)\end{array}}{\Gamma_G, \Gamma_L, \Delta \models_\tau l_{n+1}, \ldots, l_m := fp(e_1, \ldots, e_n) :: (\bullet, \mathsf{Procedure}, \Gamma_G, \Gamma_L)} \quad fp \in dom(\Gamma_G)$$

$$\frac{\begin{array}{c}\Gamma_G(fp) = ((\tau_1, \ldots, \tau_n) \rightarrow (), \mathsf{Procedure}) \\ \Gamma_G, \Gamma_L, \Delta \vdash e_1 \leq (\tau_1, \mathsf{c}_1) \quad \ldots \quad \Gamma_G, \Gamma_L, \Delta \vdash e_n \leq (\tau_n, \mathsf{c}_n)\end{array}}{\Gamma_G, \Gamma_L, \Delta \models_\tau fp(e_1, \ldots, e_n) :: (\bullet, \mathsf{Procedure}, \Gamma_G, \Gamma_L)} \quad fp \in dom(\Gamma_G)$$

$$\frac{\Gamma, \Delta \vdash e_1 \leq (\tau_1, \mathsf{cc}_1) \quad \ldots \quad \Gamma, \Delta \vdash e_n \leq (\tau_n, \mathsf{cc}_n)}{\Gamma, \Delta \models_{(\tau_1, \ldots, \tau_n)} \mathsf{return}\ e_1, \ldots, e_n :: ((\tau_1, \ldots, \tau_n), \max(\mathsf{cc}_1, \ldots, \mathsf{cc}_n), \Gamma)}$$

$$\frac{\Gamma, \Delta \models_\tau c_1 :: (\bullet, \mathsf{cc}_1, \Gamma') \quad \Gamma', \Delta \models_\tau c_2; \ldots; c_n :: (\rho, \mathsf{cc}_{2n}, \Gamma'')}{\Gamma, \Delta \models_\tau c_1; \ldots; c_n :: (\rho, \max(\mathsf{cc}_1, \mathsf{cc}_{2n}), \Gamma'')} \quad \rho \in \{\tau, \bullet\}$$

$$\frac{\Gamma, \Delta \models_\tau c_1 :: (\tau, \mathsf{cc}_1, \Gamma') \quad \Gamma', \Delta \models_\tau c_2; \ldots; c_n :: (\rho, \mathsf{cc}_{2n}, \Gamma'')}{\Gamma, \Delta \models_\tau c_1; \ldots; c_n :: (\tau, \max(\mathsf{cc}_1, \mathsf{cc}_{2n}), \Gamma'')} \quad \rho \in \{\tau, \bullet\}$$

Figure 28: Type checking rules for CAO statements (Part I).

$$\frac{\Gamma,\Delta \vdash b \leq (\mathsf{Bool},\mathsf{cb}) \quad \Gamma,\Delta \models_\tau c_1 :: (\tau,\mathsf{cc}_1,\Gamma') \quad \Gamma,\Delta \models_\tau c_2 :: (\bullet,\mathsf{cc}_2,\Gamma'')}{\Gamma,\Delta \models_\tau \mathsf{if}\ b\ \{c_1\}\ \mathsf{else}\ \{c_2\} :: (\bullet,\max(\mathsf{cb},\mathsf{cc}_1,\mathsf{cc}_2),\Gamma)}$$

$$\frac{\Gamma,\Delta \vdash b \leq (\mathsf{Bool},\mathsf{cb}) \quad \Gamma,\Delta \models_\tau c_1 :: (\bullet,\mathsf{cc}_1,\Gamma') \quad \Gamma,\Delta \models_\tau c_2 :: (\tau,\mathsf{cc}_2,\Gamma'')}{\Gamma,\Delta \models_\tau \mathsf{if}\ b\ \{c_1\}\ \mathsf{else}\ \{c_2\} :: (\bullet,\max(\mathsf{cb},\mathsf{cc}_1,\mathsf{cc}_2),\Gamma)}$$

$$\frac{\Gamma,\Delta \vdash b \leq (\mathsf{Bool},\mathsf{cb}) \quad \Gamma,\Delta \models_\tau c_1 :: (\bullet,\mathsf{cc}_1,\Gamma') \quad \Gamma,\Delta \models_\tau c_2 :: (\bullet,\mathsf{cc}_2,\Gamma'')}{\Gamma,\Delta \models_\tau \mathsf{if}\ b\ \{c_1\}\ \mathsf{else}\ \{c_2\} :: (\bullet,\max(\mathsf{cb},\mathsf{cc}_1,\mathsf{cc}_2),\Gamma)}$$

$$\frac{\Gamma,\Delta \vdash b \leq (\mathsf{Bool},\mathsf{cb}) \quad \Gamma,\Delta \models_\tau c_1 :: (\tau,\mathsf{cc}_1,\Gamma') \quad \Gamma,\Delta \models_\tau c_2 :: (\tau,\mathsf{cc}_2,\Gamma'')}{\Gamma,\Delta \models_\tau \mathsf{if}\ b\ \{c_1\}\ \mathsf{else}\ \{c_2\} :: (\tau,\max(\mathsf{cb},\mathsf{cc}_1,\mathsf{cc}_2),\Gamma)}$$

$$\frac{\Gamma,\Delta \vdash b \leq (\mathsf{Bool},\mathsf{cb}) \quad \Gamma,\Delta \models_\tau c :: (\rho,\mathsf{cc},\Gamma')}{\Gamma,\Delta \models_\tau \mathsf{if}\ b\ \{c\} :: (\bullet,\max(\mathsf{cb},\mathsf{cc}),\Gamma)} \quad \rho \in \{\tau,\bullet\}$$

$$\frac{\Gamma,\Delta \vdash b \leq (\mathsf{Bool},\mathsf{cb}) \quad \Gamma,\Delta \models_\tau c :: (\rho,\mathsf{cc},\Gamma')}{\Gamma,\Delta \models_\tau \mathsf{while}\ b\ \{c\} :: (\bullet,\max(\mathsf{cb},\mathsf{cc}),\Gamma)} \quad \rho \in \{\tau,\bullet\}$$

$$\frac{\begin{array}{c}\phi_\Delta(e_1) = i \quad \phi_\Delta(e_2) = j \quad \phi_\Delta(e_3) = k \\ \forall_{n\in\{i,i+k,\ldots,j\}}\Gamma_G,\Gamma_L[x :: \mathsf{Int}],\Delta[x := n] \models_\tau c :: (\rho,\mathsf{cc},\Gamma'_G,\Gamma'_L)\end{array}}{\begin{array}{c}\Gamma_G,\Gamma_L,\Delta \models_\tau \mathsf{seq}\ x := e_1\ \mathsf{to}\ e_2\ \mathsf{by}\ e_3\ \{\ c\ \} :: (\bullet,\mathsf{cc},\Gamma_G,\Gamma_L) \\ \rho \in \{\tau,\bullet\},\ x \notin dom(\Gamma_L),\ i \leq j,\ k \geq 1\end{array}}$$

$$\frac{\begin{array}{c}\phi_\Delta(e_1) = i \quad \phi_\Delta(e_2) = j \quad \phi_\Delta(e_3) = k \\ \forall_{n\in\{i,i-k,\ldots,j\}}\Gamma_G,\Gamma_L[x :: \mathsf{Int}],\Delta[x := n] \models_\tau c :: (\rho,\mathsf{cc},\Gamma'_G,\Gamma'_L)\end{array}}{\begin{array}{c}\Gamma_G,\Gamma_L,\Delta \models_\tau \mathsf{seq}\ x := e_1\ \mathsf{to}\ e_2\ \mathsf{by}\ e_3\ \{\ c\ \} :: (\bullet,\mathsf{cc},\Gamma_G,\Gamma_L) \\ \rho \in \{\tau,\bullet\},\ x \notin dom(\Gamma_L),\ i > j,\ k \geq 1\end{array}}$$

$$\frac{\begin{array}{c}\phi_\Delta(e_1) = i \quad \phi_\Delta(e_2) = j \\ \forall_{n\in\{i,i+1,\ldots,j\}}\Gamma_G,\Gamma_L[x :: \mathsf{Int}],\Delta[x := n] \models_\tau c :: (\rho,\mathsf{cc},\Gamma'_G,\Gamma'_L)\end{array}}{\begin{array}{c}\Gamma_G,\Gamma_L,\Delta \models_\tau \mathsf{seq}\ x := e_1\ \mathsf{to}\ e_2\ \{\ c\ \} :: (\bullet,\mathsf{cc},\Gamma_G,\Gamma_L) \\ \rho \in \{\tau,\bullet\},\ x \notin dom(\Gamma_L),\ i \leq j\end{array}}$$

$$\frac{\begin{array}{c}\phi_\Delta(e_1) = i \quad \phi_\Delta(e_2) = j \\ \forall_{n\in\{i,i-1,\ldots,j\}}\Gamma_G,\Gamma_L[x :: \mathsf{Int}],\Delta[x := n] \models_\tau c :: (\rho,\mathsf{cc},\Gamma'_G,\Gamma'_L)\end{array}}{\begin{array}{c}\Gamma_G,\Gamma_L,\Delta \models_\tau \mathsf{seq}\ x := e_1\ \mathsf{to}\ e_2\ \{\ c\ \} :: (\bullet,\mathsf{cc},\Gamma_G,\Gamma_L) \\ \rho \in \{\tau,\bullet\},\ x \notin dom(\Gamma_L),\ i > j\end{array}}$$

Figure 29: Type checking rules for CAO statements (Part II).

$$\frac{\Delta \vdash_t t_1 \rightsquigarrow \tau_1 \quad \ldots \quad \Delta \vdash_t t_n \rightsquigarrow \tau_n \quad \Delta \vdash_t t \rightsquigarrow \tau \quad \Gamma_G, \epsilon[x_1 :: \tau_1, \ldots, x_n :: \tau_n], \Delta \models_\tau c :: (\tau, \mathsf{cc}, \Gamma'_G)}{\Gamma_G, \epsilon, \Delta \models \mathsf{def}\ fp(x_1 : t_1, \ldots, x_n : t_n) : t\ \{c\} :: (\bullet, \Gamma_G[fp :: ((\tau_1, \ldots, \tau_n) \to \tau, \mathsf{cc})])}$$

where $t \neq \mathsf{void}$ and $fp \notin dom(\Gamma_G)$

$$\frac{\Delta \vdash_t t_1 \rightsquigarrow \tau_1 \quad \ldots \quad \Delta \vdash_t t_n \rightsquigarrow \tau_n \quad \Gamma_G, \epsilon[x_1 :: \tau_1, \ldots, x_n :: \tau_n], \Delta \models_{()} c; \mathsf{return}() :: ((), \mathsf{Procedure}, \Gamma'_G)}{\Gamma_G, \epsilon, \Delta \models \mathsf{def}\ fp(x_1 : t_1, \ldots, x_n : t_n) : \mathsf{void}\ \{c\} :: (\bullet, \Gamma_G[fp :: ((\tau_1, \ldots, \tau_n) \to (), \mathsf{Procedure})])}$$

where $fp \notin dom(\Gamma_G)$

$$\frac{\Delta \vdash_t t \rightsquigarrow \tau}{\Gamma, \Delta \models \mathsf{typedef}\ tid\ :=\ t :: (\bullet, \Gamma)}$$

$$\frac{\Delta \vdash_t t_1 \rightsquigarrow \tau_1 \quad \ldots \quad \Delta \vdash_t t_n \rightsquigarrow \tau_n}{\Gamma_G, \Gamma_L, \Delta \models \mathsf{typedef}\ sid := \mathsf{struct}[fi_1 : t_1; \ldots; fi_n : t_n] :: (\bullet, \Gamma_G[fi_1 :: (sid \to \tau_1, \mathsf{Pure}), \ldots, fi_n :: (sid \to \tau_n, \mathsf{Pure})], \Gamma_L)}$$

where $sid, fi_1, \ldots, fi_n \notin dom(\Gamma)$ where $fi_i \neq fi_j$ for $1 \leq i \leq n$ and $1 \leq j \leq n$

$$\frac{\epsilon, \epsilon, \epsilon \models d_1 :: (\bullet, \Gamma_{G_1}) \quad \ldots \quad \Gamma_{G_{n-1}}, \epsilon, \epsilon \models d_n :: (\bullet, \Gamma_G)}{\epsilon, \epsilon, \epsilon \models d_1; \ldots; d_n :: (\bullet, \Gamma_G)} \quad \mathsf{main} :: () \to () \in \Gamma$$

Figure 30: Typechecking rules for declarations

4

CAO-SL SPECIFICATION

CAO-SL (editor) is a specification language intended to be used in annotations of CAO programs. This specification language is strongly inspired by ACSL Baudin et al. (2010) and, similarly to ACSL, it aims to provide means that can be used to reason about behavioural properties of CAO programs. CAO-SL allows one to annotate a CAO program with pre- and postconditions, as well as the necessary invariants. Additionally, it also permits the definition of logical specifications, that can then be used in the annotation of functions.

4.1 LOGIC EXPRESSIONS

Logic expressions are a very important component of CAO-SL, since these are the expressions that are to be used in annotations. CAO-SL logic expressions correspond to boolean CAO expressions, with some additional constructors, like quantification. The grammar of logic expressions that are considered in CAO-SL are summed up in Figure 31.

The following operators were introduced:

- Connectives - the CAO boolean operators &&, || and ! are seen as logic connectives in CAO-SL. Additionally, the implication $==>$ and equivalence $<==>$ constructors were added.
- Quantification - one is able to reason about universally and existentially quantified variables in CAO-SL.
- Conditional - CAO-SL incorporates a conditional logic expression $b\ ?\ e_1\ :\ e_2$, that is equivalent to $(b \Rightarrow e_1) \wedge (!b \Rightarrow e_2)$.
- Logic functions - it is important to distinguish between logic functions and normal CAO functions. The first, are restricted to logic functions and predicates defined using CAO-SL and can not be called by CAO functions. Normal CAO functions are functions defined in the CAO program and can be called by the logic functions.
- Consecutive comparison operators - in CAO-SL, one can specify comparisons by using expressions of the form $e_1\ R_1\ e_2\ R_2\ e_3$, with $R_1, R_2 \in \{<, \leq, >, \geq, ==, !=\}$.

⟨*expr*⟩	::	**true** | **false**	boolean constants
	|	⟨*ident*⟩	identifier
	|	(⟨*expr*⟩)	
	|	⟨*expr*⟩ `?` ⟨*expr*⟩ `:` ⟨*expr*⟩	
	|	⟨*const_expr*⟩	constant expression
	|	⟨*ident*⟩ ⟨{ ⟨*label list*⟩ }⟩? `()`	function call
	|	⟨*ident*⟩ ⟨{ ⟨*label list*⟩ }⟩? `(` ⟨*expr*⟩ (,⟨*expr*⟩)* `)`	function call (arguments)
	|	⟨*expr*⟩ [⟨*expr*⟩]	
	|	⟨*expr*⟩ [⟨*expr*⟩ `,` ⟨*expr*⟩]	
	|	⟨*expr*⟩[⟨*range*⟩]	vector/bits range accesses
	|	⟨*expr*⟩[⟨*range*⟩,⟨*range*⟩]	matrix range accesses
	|	⟨*expr*⟩ ⟨*binop*⟩ ⟨*expr*⟩	
	|	⟨*unop*⟩ ⟨*expr*⟩	
	|	**old** (⟨*expr*⟩)	
	|	**result**	

⟨*range*⟩ :: ⟨*int*⟩ `..` ⟨*int*⟩

⟨*const_expr*⟩ :: ⟨*int*⟩ | ⟨*bool*⟩

⟨*binop*⟩ :: `-` | `+` | `*` | `/` | `%` | `**` | `##` | `|` | `&` | `|` | `^` | `>>` | `|>` | `<<` | `<|` | `||` | `^^`

⟨*unop*⟩ :: `!` | ∼ | `+` | `-`

⟨*relation*⟩ :: `<` | `<=` | `>` | `>=` | `==` | `!=`

⟨*predicate*⟩	::	⟨*expr*⟩	
	|	(⟨*predicate*⟩)	
	|	⟨*expr*⟩ ⟨*relation*⟩ ⟨*expr*⟩	
	|	⟨*expr*⟩ ⟨*relation*⟩ ⟨*expr*⟩ ⟨*relation*⟩ ⟨*expr*⟩	
	|	**!** ⟨*predicate*⟩	negation
	|	⟨*predicate*⟩ && ⟨*predicate*⟩	
	|	⟨*predicate*⟩ || ⟨*predicate*⟩	
	|	**forall** binders `;` ⟨*predicate*⟩	universal quantification
	|	**exists** binders `;` ⟨*predicate*⟩	existential quantification
	|	⟨*predicate*⟩ ==> ⟨*predicate*⟩	implication
	|	⟨*predicate*⟩ <==> ⟨*predicate*⟩	equivalence

Figure 31: Logic expressions grammar in CAO-SL

class	associativity	operators
unary	right	! + − &
multiplicative	left	* % ** ## @ /
additive	left	+ -
shift	left	>> <<
comparison	left	< <= > >=
comparison	left	== ! =
bitwise and	left	&
bitwise xor	left	^
bitwise or	left	|
connective and	left	&&
connective xor	left	^ ^
connective or	left	| |
connective implies	left	==>
connective equiv	left	<==>
ternary connective	right	..?.. : ..
binding	left	*forall exists*

Table 4: Operators precedence

4.1.1 *Operator precedence*

The precedence of CAO operators is maintained in CAO-SL. The precedence of the additional operators can be found in Table 4, such that operators that appear at the top of the table have higher precedence than those that appear at the top of the table.

4.1.2 *Semantics*

The semantics of the logic expressions are based on mathematical first-order logic. They are evaluated to $true$ or $false$ and functions are always total.

4.1.3 *Types in logic expressions*

CAO-SL adopts the sames types as CAO and, therefore, there is no need to add some built-in types. However, one is able to define new logic types and use them in the specification language. Note that, in CAO, the integer type already corresponds to $\mathbb{Z}$ and this is why CAO types can be adopted by CAO-SL.

4.2 FUNCTION CONTRACTS

In CAO-SL, a function contract consists exclusively of a specification of a precondition (*requires*), a postcontions (*ensures*) and the parts of the sate that are modified by the function (*assigns*). The grammar of contracts can be found in Figure 32.

$$
\begin{array}{rcl}
\langle \textit{contract} \rangle & :: & \textbf{requires}\ \langle \textit{predicate} \rangle\ \langle \textit{contract} \rangle \\
& | & \textbf{assigns nothing} \\
& | & \textbf{assigns}\ \langle \textit{location list} \rangle\ \langle \textit{contract} \rangle \\
& | & \textbf{ensures}\ \langle \textit{predicate} \rangle\ \langle \textit{contract} \rangle
\end{array}
$$

Figure 32: Grammar for function contracts in CAO-SL

A simple function contract has the form:

```
/*@
  requires P1 && ... && PN
  assigns L1, ..., LN
  ensures E1 && ... && EN
*/
```

where the order in which the clauses appear is irrelevant. The semantics of such contract is defined directly from the Hoare logic:

- $P1$ && ... && PN, as a precondition, must hold in state from which the function is being called.
- $E1$ && ... && EN, as postcondition, must hold in a state where the function returns.
- $L1, ..., LN$ must refer to global variables that are modified by the function during its execution.
- When no *requires* or *ensures* clause is provided, it will be assumed *true* as precondition or postcondition, respectively.
- The *assigns* clause can be omitted even if there are some side-effects in the function.
- In the postcondition clause, it is mandatory to reference local variables in the pre-state. One can also reason about variables inside the function boundaries using assertions (*assert*).

4.2.1 *Constructors old and result*

In most of the specifications, postconditions will refer to the *old* value of the preconditions values, i.e., to the value that they had when the function was called, or to the *result* of the function, i.e., its

output. These constructions can be found in CAO-SL: $old(e)$ denotes the value of expression e in the pre-state and $result$ denotes the value returned by the function. Note that they can only be used in postcondition (*ensure*) clause.

4.2.2 *State and locations*

Locations are the parts of the state that are modified by the function. As presented previously, one can define the set of locations of some block of code using the *assigns* clause. Therefore, a location can be seen as set of expressions that denote the modifiable left-values. These left-values can be specified in CAO-SL the same way they are written in CAO code and their grammar can be found in Figure 18 (*LValues* syntactic domain).

4.3 STATEMENT ANNOTATIONS

CAO-SL allows two type of statement annotations:

- Assertions - that can be introduced before any CAO statement and after block statements.
- Loop annotations - that are allowed before any loop statement.

4.3.1 *Assertions*

An assertion, of the form *assert* p, means that the predicate p must hold in the current state (typically before some CAO statement). When an assertion is proved, it becomes part of the context and can be used to prove the subsequent proof obligations of the proof tree. The grammar for assertions is given in Figure 33.

$$\langle statement \rangle \quad :: \quad \langle assertion \rangle \ \langle statement \rangle$$

$$\langle assertion \rangle \quad :: \quad /\star @ \ \textbf{assert} \ \langle formula \rangle \ \star /$$

Figure 33: Grammar for assertions in CAO-SL

4.3.2 *Loop annotations*

Loops may be annotated with invariant and variant expressions and with some *assigns* clause. The grammar for loop annotations can be found in Figure 34.

LOOP INVARIANTS A loop invariant annotation has the following form:

⟨*statement*⟩ :: ⟨*loop_annotation*⟩*
while (⟨*expr*⟩) ⟨*statement*⟩
| ⟨*loop_annotation*⟩*
seq ⟨*ident*⟩ **:=** ⟨*expr*⟩ **to** ⟨*expr*⟩ ⟨*by_node*⟩? ⟨*statement*⟩

⟨*loop_annotation*⟩ :: /*@ ⟨*node*⟩* */

⟨*node*⟩ :: **invariant** ⟨*formula*⟩
| **assigns** ⟨*location*⟩
| **variant** ⟨*formula*⟩

⟨*assigment*⟩ :: ⟨*expr*⟩(, ⟨*expr*⟩)* **:=** ⟨*expr*⟩(, ⟨*expr*⟩)*

⟨*by_node*⟩ :: **by** ⟨*expr*⟩

Figure 34: Grammar for loop annotations in CAO-SL

```
/*@ invariant I
   assigns L */
```

We recall the semantics of loop invariants, already mentioned in Section 2.3. Informally, the predicate I must hold before, during and after the execution of the loop. More formally, I is an inductive invariant. Consequently, it must hold for some base case (before the execution of the loop) and, if I is assumed true in some state where the loop condition holds, and if the execution of the loop does not abruptly terminate, I is true in the resulting state.

The *assigns* clause for loop annotations has the same semantic as the *assigns* clause for function contracts.

LOOP VARIANTS A loop variant is a condition that is used to prove that the loop, in fact, terminates. It has the form $variant\ V$, where V must be a non-negative integer expression.

The semantics of this construction expresses the fact that, for each loop iteration that does not terminates abruptly, the value V must decrease. At some point, the value of the variant will match to the value of the loop condition and termination for the given loop will be proved.

CONSTRUCTOR at When annotating loops, it is sometimes necessary to refer to the value of some expression in some particular state. We have seen that CAO-SL contemplates the use of the constructors old and $result$ in function contracts. In loop annotations, one can use the constructor $at(e, id)$, that denotes to the value of the expression e in some state labeled as id.

CAO-SL allows the declaration of labels as a statement annotation $/*@label\ id*/$. Nevertheless, in CAO-SL there are four predefined logic labels, that can only be used inside function contracts and function statement annotations:

- **Here** - visible in all statement annotations and it refers to the current state where an annotation appears. It is also visible in function contracts: in **requires**, it refers to the pre-state, whereas in **assigns** and **ensures** it refers to the post-state.
- **Old** - visible in assings and **ensures** clauses and refers to the pre-state of the function. It is equivalent to write **old**(e) and **at**(e,**Old**).
- **Pre** - visible in all statement annotations and refers to the pre-state of the function.
- **Post** - visible in **assigns** and **ensures** clauses and refers to the post-state of the function.

4.4 LOGIC SPECIFICATIONS

CAO-SL supports the definiton of logic specifications which allows the declaration of new types, logic functions, predicates and lemmas. The grammar for these declarations is presented in Figure 35.

⟨*logic-decl*⟩	::	**predicate** ⟨*ident*⟩ ⟨*label-decl*⟩?	predicate
		(⟨*parameters*⟩?) = ⟨*predicate*⟩ ;	
	|	**logic** ⟨*type*⟩ ⟨*ident*⟩ ⟨*label-decl*⟩?	logic function
		(⟨*parameters*⟩?) = ⟨*expr*⟩ ;	
	|	**lemma** ⟨*ident*⟩ ⟨*label-decl*⟩? : ⟨*predicate*⟩	lemma
	|	**logic def** ⟨*ident*⟩ : ⟨*type*⟩ ;	logic variable
	|	**logic** ⟨*ident*⟩ ;	logic type
⟨*label-decl*⟩	::	{ ⟨*label list*⟩ }	

Figure 35: Grammar for logic specifications in CAO-SL

4.4.1 *Functions*

Logic functions can be defined using the keyword $logic$, preciding its return type and its definition. For example, a logic function that computes the result of the addition of two values a and b can be specified as follows

```
/*@ logic int sum{L} (a,b : int) = a + b; */
```

4.4.2 *Predicates*

A predicate can be seen as function that evaluates in some boolean value. One example of a predicate is the following

```
/*@ predicate equal{L1,L2}(u,v : unsigned bits[10]) =
    forall l:int; 0 <= l < 10 ==> at(u[l],L1) == at(v[l],L2); */
```

where $equal$ tests if two vectors are equal in two different states $L1$ and $L2$.

INDUCTIVE PREDICATES One can also define inductive predicates in CAO-SL, like shown in Figure 36. For a better understanding of the semantics of a logic predicate, consider the following example

⟨*inductive_predicate*⟩ :: **inductive** ⟨*ident*⟩ ⟨*label_decl*⟩? (⟨*parameters*⟩?)
{ **case*** }

⟨*case*⟩ :: `case` ⟨*ident*⟩ ⟨*label_decl*⟩? **:** ⟨*predicate*⟩

Figure 36: Grammar for inductive predicates in CAO-SL

```
/*@ inductive isfib(n,r : int) {
     case isfib_zero : isfib(0,0);
     case isfib_one : isfib(1,1);
     case isfib_ind :
       forall n,r1,r2 : int;
         isfib(n,r1) && isfib(n+1,r2) ==> isfib(n+2,r1+r2);
} */
```

that defines the Fibonacci property. This inductive definition expresses that, for zero and one (base cases), the Fibonacci number is always the same and, for values greater than one (inductive cases), the computation of the Fibonacci number is based on two immediatly prior iterations.

4.4.3 *Lemmas*

Lemmas are prepositions defined with the intent to make it easier to prove the validity of some specification. Note that, contrarly to axioms, lemmas need to be proved and, therefore, a proof obligation related to it is also generated. A lemma can be defined as in the following example, that represents the transitivity relation for equality over unsigned bit strings.

```
/*@ lemma transitivity{L1,L2,L3}:
     forall u,v,z: unsigned bits[10];
       equal{L1,L2}(u,v) && equal{L2,L3}(v,z) ==>
       equal{L1,L3}(u,z); */
```

4.4.4 *Axiomatic definitions*

An alternative way to define predicates, logic functions and logic types is by the use of $axiomatics$. An $axiomatic$ is a logic definition that contemplates logic types, predicates and operators over those types. Figure 37 shows the grammar rules to specifiy these kinds of annotations. An example of an axiomatic follows.

```
/*@ axiomatic lists_axiomatic {
    logic list;
    logic def nil:list;
    logic list append(l1:list, l2:list);
    logic list cons(n:int, l:list);
    axiom append_nil{L}: forall l:list; append(l,nil) == l;
    axiom append_cons{L}: forall l1,l2:list, n:int;
      append(cons(n,l1),l2) == cons(n,append(l1,l2));
} */
```

In this example, there is the definition of a logic function $append$, that concatenates two lists. We recall that axioms do not need to be proved, as their are assumed true, and, thus, a new proof goal is not generated for them.

⟨*axiomatic_decl*⟩ :: **axiomatic** ⟨*ident*⟩ { ⟨*axiomatic_pred list*⟩ }

⟨*axiomatic_pred*⟩ ::
- **axiom** ⟨*ident*⟩ ⟨*label-decl*⟩? **:** ⟨*predicate*⟩ **;**
- **predicate** ⟨*ident*⟩ ⟨*label-decl*⟩? (⟨*parameters*⟩?) (= ⟨*predicate*⟩)? **;**
- **logic** ⟨*type*⟩ ⟨*ident*⟩ ⟨*label-decl*⟩? (⟨*parameters*⟩?) (= ⟨*expr*⟩)? **;**
- **logic** `def` ⟨*ident*⟩ **:** ⟨*type*⟩ **;**
- **logic** ⟨*ident*⟩ **;**

Figure 37: Grammar for axiomatics in CAO-SL

4.5 GHOST CODE

Ghost code Filliâtre et al. (2014) is a very powerfull tool that helps validating proof obligations. This type of code is invisible to the compiler and does not interfere with the actual code written. However, it can enhance the code with additional information that help proving some property of the code by only being visible to the proof tool.

Ghost code, in CAO-SL, is similar to CAO code (in fact, one can define any CAO statement in ghost code), with the subtlety of being inside annotations and to start with the keyword $ghost$. The grammar for ghost code in CAO-SL can be found in Figure 38.

There are some important aspects that need to be taken into account when dealing with ghost code. Ghost code must not interfere with the memory and the states of the CAO program and the locations of the ghost code must be disjoint form the ones associated with the real program. Therefore, every execution of ghost code - even if some non-ghost block of code is called - must not change any CAO variable or struct field. Summing up, the control-flow graph of a CAO function must not be altered by ghost code.

⟨*ghost-type*⟩	::	⟨*CAO-type*⟩ | logic-type	types declaration
⟨*declaration*⟩	::	⟨*CAO-declaration*⟩	
	|	/*@ **ghost** ⟨*ghost-decl*⟩ */	ghost declarations
⟨*statement*⟩	::	⟨*CAO-statement*⟩	
	|	/*@ **ghost** ⟨*ghost-statement*⟩+ */	ghost statements
⟨*ghost-selection-statement*⟩	::	⟨*CAO-selection-statement*⟩	
	|	**if** (⟨*expr*⟩) ⟨*statement*⟩	
		/*@ **ghost else** ⟨*CAO-statement*⟩+ */	extended if-else
⟨*struct-declaration*⟩	::	⟨*CAO-struct-declaration*⟩	
	|	/*@ **ghost** ⟨*CAO-struct-declaration*⟩+ */	ghost fields

Figure 38: Grammar for ghost code in CAO-SL

5

EASYCRYPT TOOLSET

EasyCrypt Barthe et al. (2011a) is a toolset for reasoning about relational properties of probabilistic computations with adversarial code. Its main application is the construction and verification of game-based cryptographic proofs. EasyCrypt allows one to formalize types, operators, distributions, algebraic properties, schemes, parametric games and adversaries in order to develop cryptographic proofs.

We already described the tool, informally and with few details, in Section 1.4. In this chapter, we provide a more detailed description of EasyCrypt, showing its possible applications, as well as a typical usage of EasyCrypt when performing proofs.

5.1 AN EXAMPLE OF EASYCRYPT

Consider the Bellare and Rogaway Bellare and Rogaway (1994) (BR95) encryption scheme. Let $\mathcal{M}$ be the type of message and $\mathcal{R}$ the type of randomness. Let $(\mathcal{K}_f, f, f^{-1})$ be a family of trapdoor permutations on $\mathcal{R}$ and $G : \mathcal{R} \to \mathcal{M}$ a hash function. The BR95 scheme is composed of:

$$\begin{gathered}\text{kg}() = (pk, sk) \leftarrow \$\mathcal{K}_f; \text{return } (pk, sk)\\ \text{enc}(pk, m) = r \leftarrow \$\mathcal{R}; \text{return } (f\ pk\ r, m \oplus G\ r)\\ \text{dec}(sk, c) = (s, t) = c; r = f^{-1}\ sk\ s; \text{return } t \oplus G\ r\end{gathered}$$

Figure 39: BR95 encryption scheme

SPECIFICATION OF BASIC TYPES AND OPERATIONS A typical proof in EasyCrypt starts by defining the types that will be involved in the definition of the scheme and in the desired proof. EasyCrypt comes with a set of theories that already define some basic types like integers, booleans or bit strings. However, one is always able to define new types to help structuring proofs.

In the case of the BR95 encryption scheme, and analysing its specification in Figure 39, there is no concrete definition of what the sets $\mathcal{K}$, $\mathcal{M}$ and $\mathcal{R}$ are. In the EasyCrypt model of the scheme, we keep these types abstract also. One can declare types and operators like:

```
type pkey, skey.
```

```
type keys = pkey * skey.

type plain.
type rand.
type cipher = rand * plain.

op zero : plain.
op ( + ) : plain → plain → plain.

op G : rand → plain.
```

EasyCrypt also allows one to deal with polymorphic types.

```
type α list.

op ( :: ) : α → α list → α list.
```

The syntax $(\_)$ defines an infix operation.

PROPERTIES OF BASIC OPERATIONS We have defined in the previous section a new type - *plain* -, an operation over that type - $(+)$ - and an element which value is of type *plain* - *zero*. In order to the cryptosystem to be correct, the operation $(+)$ needs to have a certain behaviour, i.e., needs to behave like the *xor* operation. The properties of the *xor* operation are well known:

- Identity element - $\exists y : plain, \forall x : plain, x + y = x.$
- Commutativity - $forallx, y : plain, x + y = y + x.$
- Associativity - $\forall x, y, z : plain, x + (y + z) = (x + y) + z.$
- Cancelation - $\forall x : plain, x + x = zero.$

One can specify this behaviour through a set of axioms, like described bellow.

```
axiom xor0p (x:plain): zero + x = x.

axiom xorC (x y:plain): x + y = y + x.

axiom xorA x y z : x + (y + z) = (x + y) + z.

axiom xorN x : x + x = zero.
```

After, one can start by performing some proofs over the operation.

```
lemma xorp0 x : x + zero = x.
proof.
  rewrite xorC.
  apply xor0p.
qed.

lemma xorAN x y : (x + y) + y = x by smt.
```

In the example above, we prove simple properties, that combine the axioms already defined. The first lemma combines the $xor0p$ and $xorC$ axioms and the second lemma combines the $xorA$ and $xorN$ axioms.

Additionally, we have introduced two ways of dealing with proofs: either by using a tactic language - similar to COQ- and perform the proof interactively or by using SMT solvers (tactic *smt*) to discharge proofs. Note that, in a more complex proof script, one is always able to combine the two styles.

SPECIFICATION OF RANDOM OPERATORS Random operators are of extreme importance to cryptography. They allow the definition of random samplings from probabilistic distributions and are the core of key generation algorithms. Random operators are defined using a special polymorphic type - *distr* - and an operator *mu* - which returns the probability of an event in a given discret sub-distribution. The *mu* operator was briefly introduced in Section 2.4 as the μ operator. The following example is extracted from the EasyCrypt *Distr* theory.

```
type α distr.

op mu : α distr → (α → bool) → real.
```

As mentioned in Section 2.4, the operator **mu d E** should be understood as

$$\Pr[x = \$d \ : \ E\ x]$$

For the BR95 encryption scheme, on can specify the random operators as follows

```
op drand : rand distr.

axiom drand_lossless : mu drand True = 1.

axiom drand_uniform : is_uniform drand.
```

The predicate $is\_uniform$ attests if a some probabilistic distribution is uniform (the probabilities of the possbile values in it are the same).

SPECIFICATION OF f AND f^{-1} To complete the formalisation of the parameters of the BR95 encryption scheme, it is missing the declaration of the functions f and f^{-1}. One possible formalisation of these components is the following:

```
op keygen : keys distr.

op f : pkey → rand → rand.
op finv : skey → rand → rand.

axiom finvof pk sk x: in_supp (pk,sk) keygen ⇒ finv sk (f pk x) = x.

axiom fofinv pk sk x: in_supp (pk,sk) keygen ⇒ f pk (finv sk x) = x.
```

Again, there is no explicit realisation of function f or f^{-1}. Its behaviour is modeled according to the two axioms $finvof$ and $fofinv$, that act like cancelation axioms and that, informally, specify that one function is the inverse of the other.

THE BR95 ENCRYPTION SCHEME The formalisation of the BR95 encryption scheme can be done using an EasyCrypt *module*. We have already explained with little detail this component of the toolset in Section 1.4. Modules are a keystone of EasyCrypt. They allow the specification - using the *pWhile* language - of encryption schemes, oracles, adversaries, cryptographic assumptions or game-based properties, in the form of probabilistic programs.

```
module BR95 = {
  proc kg() : keys = { var ks : keys; ks = $keygen; return ks; }

  proc enc(pk:pkey, m:plain) : cipher = {
    var r : rand;

    r = $drand;

    return (f pk r, m + G r);
  }

  proc dec(sk:skey, c:cipher) : plain = {
    var s : rand;
    var t : plain;
```

```
    (s,t) = c;

    return (t + G (finv sk s));
  }
}.
```

The definition presented above declares a *module* BR95 with three procedures (*kg*, *enc* and *dec*). One is also able to define global variables on modules, call external modules, parametrize modules by other modules, define high order modules, perform quantification - *forall* and *exists* - and to define restrictions on which procedures from a *module* can be called.

5.2 PROVING IN EASYCRYPT

We have previously mentioned that one can follow two *styles* when dealing with formal verification of programs: one with more focus on automation and another one with more focus on interactivity. The first one relies on automated external proof tools that try to discharge proof obligations without intervention of the user. Clearly, the work is *limited* to the correct annotation of the program and then the calling of external provers. However, it may be the case that a program and its respective annotations create a valid Hoare triple but the external provers are not able to prove this property, for example, due to time requirements or ambiguity in some proof goal that prevents the tool from continuing. This last difficulty can be completely overcame with the use of an interactive proof assistant like COQ. Even if there are two or more tactics that would suit the proof goal, it is the user that chooses which one is to be applied. One can also argue that, by using an interactive proof assistant, one is able to perform a larger set of proofs than using an SMT solver. Nevertheless, the degree of automation in an interactive proof assistant is very limited and, for some proof goal, a call to an SMT solver would close it but, following the nature of interact proofs, one needs to perform it all.

In this section we introduce how EasyCrypt can be used to perform proofs over programs. EasyCrypt is supported by two logics: an ambient logic to reason about logic operators and first-order logic predicates and a probabilistic relational Hoare logic (which *includes* the probabilistic relational Hoare logic and the Hoare logic) to reason about properties over programs. Informally, the objective is to, using the probabilistic relational Hoare logic, transform some Hoare triple into a first-order logic goal and then discharge it using the ambient logic. Therefore, EasyCrypt introduces a new style in program verification - a semi-automated style - where the user is always free to call an SMT solver to discharge proofs, instead of performing them step-by-step. Naturally, some SMT calls will not be able to discharge the proof goal (due to its complexity or due to lack of context), which would require the user to refine the it - using the EasyCrypt logics - in order to be able to call the external prover again and succeed with the proof. In what follows this section, we will describe how EasyCrypt can be used to perform deductive program verification, by showing its proof engine, some important tactics

and an example of a correctness proof. For a more complete description of the tool, we refer to the EasyCrypt Reference Manual The EasyCrypt development team (2015).

5.2.1 *Proof engine*

Judgments in EasyCrypt are of the form

$$\epsilon; \Gamma \vdash \phi$$

where ϵ is the global environment, Γ is the context and ϕ is the proof objective. The global environment ϵ is composed of EasyCrypt theories, like the integer theory or the array theory. The context Γ is a set of assertions that are known to be true and are used to prove the goal ϕ. Typically, the interaction between the context and the proof goal works by implication: being $ass(\Gamma)$ the set of assertions of Γ, $ass(\Gamma) \Rightarrow \phi$. For example, the judgement

$$Int; x, y, z : int, x \leq y \vdash x + z \leq y + z$$

states that in the global environment solely composed of the theory of integers (Int), having three local variables x, y and z of type int along with the fact that $x \leq y$ (the context Γ), we are interested in proving $x + z \leq y + z$.

Additionally, a set of deduction axioms is given. These deduction rules are the same as described in Chapter 2 and have the form

$$\frac{P_1, ..., P_n}{\epsilon; \Gamma \vdash \phi}$$

which declares that if $P_1, ..., P_n$ are derivable, then $\epsilon; \Gamma \vdash \phi$ is also derivable.

We have also previously mentioned that the objective, when performing proofs, is to build a tree rooted by some judgement and with leaves composed of deduction rules. If a tree can be built for some judgement, then it holds in respect to the premises. The EasyCrypt proof engine helps the user to build such proofs. At each step of the proof construction, the system presents to the user the set of goals that have to be proved. The user can then apply a tactic to one of them, each tactic corresponding to a deduction rule. If the conclusion of the rule corresponding to the applied tactic matches the goal to which it is applied, the proof engine replaces it with the set of the premises of the applied rule - the subgoals. This application may generate zero, one or several subgoals depending on the rule. This process is repeated iteratively, up to the point where no goals remain. At this point, the proof is closed.

5.2.2 *Ambient logic*

Ambient logic is used to reason about first-order logic proof goals. The tactics that support it are similar to COQ tactics that provide means to reason with propositional and first order logic lemmas. We provide description of some important tactics of the ambient logic.

`MOVE =>` $\pi_1 \ldots \pi_n$ | `MOVE :` $\pi_1 \ldots \pi_n$ Moves the assumptions $\pi_1 \ldots \pi_n$ from the proof goal to the context in the first case and from the context to the proof goal in the second case.

`APPLY` p Considering the formula $\forall (x_1 : \tau_1) \ldots (x_n : \tau_n), P_1 \Rightarrow \ldots \Rightarrow P_n \Rightarrow G$, `apply` p tries to match G with the representation of p. If it succeeds, it generates n subgoals $P_1, \ldots, P_n$.

`ASSUMPTION` Search in the context for an hypothesis that is convertible to the goal and applies it.

One can also use `exact` p to refer to the exact hypothesis that is to be applied.

`CUT` $id : \phi$ Logical cut. Generates two subgoals: one for the cut formula ϕ, and one for $\phi \Rightarrow G$ where G is the current goal.

`REWRITE` $\pi_1 \ldots \pi_n$ Rewrites the rewrite-pattern $\pi_1 \ldots \pi_n$ from left to right, where the π_i are proof-terms.

`SPLIT` Breaks an intrinsically conjunctive goal into its component subgoals.

`LEFT` Reduces a disjunctive goal to its left member.

`RIGHT` Reduces a disjunctive goal to its right member.

`ELIM` $/\phi$ $\pi_1 .. \pi_n$ Applies the elimination principle ϕ to the top assumption after having generalised $\pi_1 .. \pi_n$.

`REFLEXIVITY` Solves goals of the form $t = t$.

`TRIVIAL` Tries to solve the goal by using a mixture of low-level tactics.

`SMT` Tries to solve the goal using SMT solvers. The goal is sent along with the local hypotheses plus selected axioms and lemmas.

5.2.3 *Program logics*

Program logics are to be applied when the goal assumes the shape of a Hoare triple, wether one is dealing with a Hoare logic judgement (Section 2.3), a probabilistic Hoare logic judgement (Section 2.4) or a probabilistic relational Hoare logic judgement (Section 2.5).

Using program logics, EasyCrypt allows the usage of three classes of tactics: those that operate at the level of specifications - strengthening, combining or splitting goals without modifying the program -, those that actually reason about the program in Hoare logic style and those that correspond to semantics-preserving program transformations or compiler optimisations. In this section, we will present a set of relevant tactics that fit in these classes.

5.2.3.1 *Reasoning about specifications*

`CONSEQ (_ : ` $\phi \Rightarrow \psi$ `)` Applies the $conseq$ rule.

`CASE` ϕ Split the current goal by doing a case analysis in the precondition.

5.2.3.2 *Reasoning about programs*

The tactics presented in this section have been introduced when the derivation systems for Hoare logic, probabilistic Hoare logic and probabilistic relational Hoare logic were explained. Most of the tactics introduced bellow have the behaviour of the rules showed in Chapter 2.

`PROC` Derive a specification for a concrete procedure from a specification on its code.

`INLINE` p When a given program has a procedure call to procedure P, `inline` P expands the code of P replacing it by its invokation.

`SKIP` Applies the $skip$ rule.

`SEQ` θ Applies the seq rule, using θ as the middle assertion.

`WP` Computes the weakest precondition of a straightline deterministic suffix of the program(s) that implies the current postcondition. Also consumes deterministic if statements (when both branches are deterministic straightline code without procedure calls). The `wp` tactic also consumes $assertion$ commands.

`RND` Applies the rnd tactic.

`IF` Applies the if tactic. Only works if the condition is in the first line of the program.

`WHILE` i | `WHILE` $i\ v$ Applies the $while$ tactic using i as the loop invariant in the first case and applies the $while$ tactic using i as the loop invariant and v as loop variant in the second case.

`CALL (_ :` $\phi \Rightarrow \psi$`)` | `CALL (_ :` i`)` Applies the $call$ tactic using $\phi \Rightarrow \psi$ as specification for the procedure to be called in the first case and applies the $call$ tactic using I as invariant of the procedure to be called.

5.2.3.3 *Transforming programs*

`SWAP` $l_1\ l_2\ l_3$ Swaps the code between lines l_1 and l_2 with the code between lines l_2 and l_3, assuming that these are syntactically independent.

5.2.3.4 *Automated tactics*

EasyCrypt also provides a set of tactics that automate the process of reasoning about program instructions. We provide explanation of two of those tactics.

`AUTO` Tries to apply, as much as possible, the program tactics described above. If it founds some instruction that introduces ambiguity, it stops.

`SIM` Only works for probabilistic relational Hoare logic goals. *Simulates* the execution of the two programs being compared. If the executions are the same, it discharges all the associated proof goals.

5.2.4 *A proof example: Correctness of BR93*

In this section we provide an example of how EasyCrypt can be used to perform proofs over (probabilistic) programs. We exemplify it by showing two possible ways of performing a correctness proof for the BR93 encryption scheme: one without using automatic tactics and SMT calls and another with great focus on automation on the built of the proof tree.

The first step is to write a new module $Correctness$, that is parameterised by a cryptographic encryption scheme and that executes by initialising the scheme with the key generation, encrypts a given message and then decrypts the resulting ciphertext. It returns a boolean, stating that the result of the decryption algorithm should be equal to the original message.

```
module Correctness(S: Scheme) = {
  proc main (m: plain) : bool = {
    var sk : skey;
    var pk : pkey;
    var c : cipher;
    var d : plain;
```

```
    (pk,sk) = S.kg();
    c = S.enc(pk,m);
    d = S.dec(sk,c);

    return (m = d);
  }
}.
```

$Scheme$ is a $module\ type$, i.e., is a module that only contains the header of the functions, instead of its explicit definitions, and that defines a class that other modules can inherit. For example, the module $BR93$ inherits $Scheme$, meaning that it will contain procedures with the exact same header of the ones defined in the module type.

```
module type Scheme = {
  proc kg() : keys
  proc enc(pk:pkey, m:plain) : cipher
  proc dec(sk:skey, c:cipher) : plain
}.
```

The correctness of the scheme can, at this point, be modeled as a simple Hoare triple, of the form

$$\{true\}Correctness(BR93).main\{res\}$$

where res is the result of the function, and it is written in EasyCrypt as follows.

```
lemma Correctness : hoare [Correctness(BR93).main : true ==> res].
```

PROVING CORRECTNESS INTERACTIVELY The proof is developed in two steps:

1. Apply program tactics to reason about each instruction, until reaching the empty program, where *skip* tactic transforms it in an first-order logic implication, between the precondition and the postcondition.

2. First-order logic reason, to prove that the implication hold and, consequently, prove the validity of the Hoare triple.

The proof is presented bellow.

```
lemma Correctness_interact : hoare [Correctness(BR93).main : true ⇛ res].
```

```
proof.
  proc ⇒ //.
  inline∗.
  do(wp;rnd).
  skip.
  move ⇒ &hr HT ks Hks r Hr.
  rewrite finvof.
  cut → : (ks.‘1, ks.‘2) = (fst ks, snd ks).
    rewrite /fst /snd.
    by reflexivity.
  rewrite  pairS.
  by exact Hks.
  rewrite xorAN.
  by reflexivity.
qed.
```

PROVING CORRECTNESS WITH FOCUS ON AUTOMATION The same proof can be done with a one-line proof script if one takes advantage of the full capabilities of EasyCrypt. Since we are dealing with a program made of simple instructions - mostly random samplings and assignments - one can apply the *auto* tactic and avoid the instruction-by-instruction analysis of the program. At the end, instead of developing a first-order logic proof, an external solver can be called to discharge the resulting proof goal. The proof script is presented bellow.

```
lemma Correctness_auto : hoare [Correctness(BR93).main : true ⇛ res].
proof.
  by proc; inline∗; auto; smt.
qed.
```

6

A NEW CAOVERIF

The CAO language allows the extraction of highly efficient C code, that corresponds directly to some implementation of a cryptographic primitive made in CAO. Therefore, one could use state-of-art tools to reason about the C code generated. For example, some user could use the Frama-C platform, using the Jessie or WP, to perform deductive verification or even use some bounded model checker to verify more lightweight properties.

However, analysing the C code extracted from a CAO implementation can be a very hard process. Consider, for example, that one has no control over the code generated or what data structures will be used. The solution is, thus, to perform verification over the CAO code and then generate C code that is correct by construction. This was the aim of CAOVerif Almeida et al. (2014): to develop a domain-specific deductive verification tool, allowing for the same verification techniques and with a high degree of automation.

CAOVerif follows the same approach used in other scenarios for general-purpose languages such as Java and C Filliâtre and Marché (2007). The tool architecture itself fundamentally relies on the Jessie plug-in, which itself uses Why as a back-end and is one of the components integrated into the Frama-C framework. The workflow of CAOVerif is the following:

1. An annotated CAO program (which can be processed without change by the CAO compiler, since annotations are included in the code as comments) is first checked for syntactic and typing errors. The resulting Abstract Syntax Tree (AST) is then translated into Jessie input language.

2. Most of the CAO types are not Jessie native types (e.g. extension fields, bitstrings, etc), thus the translation includes the axiomatic model of the CAO type system in first-order logic plus the translation of the CAO annotated program.

3. The proof obligations are generated by running the Jessie plug-in, which uses Why as a back-end. The proof obligations can then be discharged with some existenting automatic prover or proof assistant.

In this chapter, we describe a new CAOVerif tool, that relies on EasyCrypt as a backend. When CAOVerif was developed, there was no tool specific to the domain of cryptography that could be used as a backend for CAOVerif and, consequently, the tool needed to rely on state-of-the-art tools that

did not aimed to provide means to reason about cryptographic properties. With the development of EasyCrypt, it makes sense to evaluate its behaviour as a backend for a cryptographic domain specific deductive verification tool.

6.1 A NEW ARCHITECTURE FOR CAOVERIF

The principle behind the conception of the new CAOVerif tool is the same of the previous version. An annotated CAO program is first analysed by our implementation of the typechecker but, instead of translating it into a Jessie input language program, we produce an EasyCrypt script that matches the original specification of the CAO program.

In the old CAOVerif version, most of the CAO types were not Jessie native types. In fact, only the integers and booleans CAO types could be directly mapped into the integers and booleans Jessie types. However, since EasyCrypt is a toolset specific to the domain of cryptography, it already has the definition of important cryptographic types. Therefore, our formalisation of the CAO types relies almost all in the formalisation of EasyCrypt types.

EasyCrypt does not provide the same degree of automation that one was able to achieve by relying on the Frama-C/Jessie toolchain. More precisely, we will not be able to match the automatic generation of proof obligations feature of the old CAOVerif, neither the convenience of being able to discharge the generated proof goals with some existent SMT solver or proof assistant in a graphical way. Nevertheless, one can argue that the architectures of both versions of CAOVerif are the same, considering that EasyCrypt uses the Why3 Bobot et al. (2011); Filliâtre and Paskevich (2013) as backend in order to communicated with automatic provers.

A graphical description of architecture of the new CAOVerif tool can be found in Figure 40.

6.2 AN OCAML IMPLEMENTATION OF THE CAO TYPECHECKER

As one is able to observe in Figure 40, the first step of the CAOVerif toolchain is to parse the CAO file and to typecheck it. Since the new CAOVerif tool relies on EasyCrypt (that was developed in OCaml) the first step when developing the new CAOVerif was to implement the typechecker in the OCaml language, so that the interoperability between the two platforms would be potentialised to the maximum.

The main purpose of the CAO typechecker is to analyse the code in such a way that the code transformation and C code generation is optimised. Providing that the typechecker was to be implemented from scratch and that the aim of the CAOVerif tool is to formally verify CAO programs, our OCaml implementation of the CAO typechecker was developed with the objective of maximising the efficiency of the typechecker complex verifications (like array accesses) and of generating an AST from which the generation of an EasyCrypt script would be a direct translation.

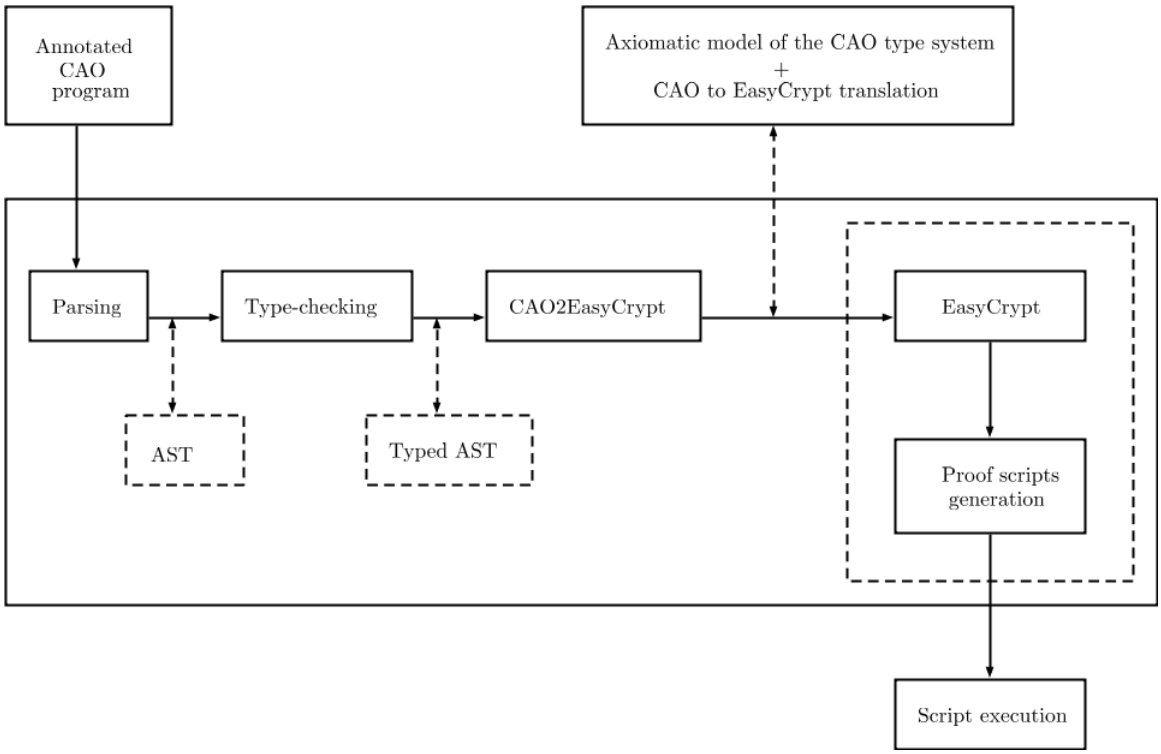


Figure 40: New CAOVerif architecture

In what remains this thesis, everytime we refer to CAO, we will be referring to the extended version of the language that we present in this section.

6.2.1 *CAO + CAO-SL: a new language*

The typical approach when building deductive verification platforms similar to CAOVerif is to develop an annotation language - that is to be written inside comments - such that this language is only readable by the deductive verification tool. This way, there is always some degree of independency between the language and all the toolset associated to it and the verification tool. In this project, we gathered the CAO language and the CAO-SL language together, resulting in a new language - that supports program descriptions and annotations - and in a new toolset - that allows one to use the capabilities of the CAO compiler along with the capabilities of the verification tool we present.

A natural implication of the combination of CAO and CAO-SL is the extension of the syntactic domains of the language. A CAO program can now be composed by global variables declarations, new types declarations, functions and procedures declarations and by logic specifications, axiomatic definitions, global invariants specifications and by ghost code definition. The grammar and syntax rules for CAO-SL elements can be found in Chapter 3.

$$
\begin{array}{rcl}
pg & :: & \ldots \\
 & & \mid ld \\
 & & \mid ad \\
 & & \mid gi \\
 & & \mid gc \\
 & & \\
 & & \ldots
\end{array}
$$

Functions and procedures now support the specification of contracts. Therefore, the syntax is extended as follows.

$$dfp \quad :: \quad [ctr]\ \mathbf{def}\ fp\ (x_1 : t_1, \ldots, x_n : t_n) : rt\ \{\ c_1; \ldots; c_m\ \}$$

The statements syntactic domain also gets extended in order to support statement annotations - loop invariants and assertions.

$$
\begin{array}{rcl}
c & :: & \ldots \\
 & & \mid \mathbf{assert}(e) \\
 & & \mid [linv]\ \mathbf{while}\ (b)\ c \\
 & & \mid [linv]\ \mathbf{seq}\ x := e_1\ \mathbf{to}\ e_2\ c \\
 & & \mid [linv]\ \mathbf{seq}\ x := e_1\ \mathbf{to}\ e_2\ \mathbf{by}\ e_3 c \\
 & & \ldots
\end{array}
$$

Finally, expressions are also extended in order to support expressions defined by the *at*, *old* and *result* constructors and by logic functions calls.

$$
\begin{array}{rcl}
e & :: & \ldots \\
 & & \mid \mathbf{old}(e) \\
 & & \mid \mathbf{result} \\
 & & \mid \mathbf{at}(la, e) \\
 & & \mid lf(e_1, \ldots, e_n) \\
 & & \ldots
\end{array}
$$

6.2.2 *Additions to the CAO language*

Besides the available commands and operators, the CAO language (and its typesystem) was extended with two additional features, that are important to cryptographic implementations.

SAMPLING OPERATOR The CAO language presented in Chapter 3 does not contemplates a sampling operation. This limitation of the language makes impossible, for example, the complete specification of key generation algorithms and, consequently, the formal definition of a cryptographic encryption scheme (which contemplates always three algorithms - a key generation algorithm, an encryption algorithm and a decryption algorithm).

The syntax of the language is, therefore, extended as shown below.

$$\begin{array}{rcl} c & := & \ldots \\ & & \mid l := \$ \\ & & \ldots \end{array}$$

The new statement means that a value of the type of l will be sampled and stored in l. For example, the following function body

```
def r : mod[7];

r := $;

return r;
```

will return a random value of the field $\mathbb{Z}_7$, i.e., a value between 0 and 6.

The corresponding typechecking rule can be found in Figure 41.

$$\frac{\Gamma, \Delta \vdash l :: (\tau, cl)}{\Gamma, \Delta \models_{\rho} l := \$ :: (\bullet, cl)}$$

Figure 41: Typechecking rule for the sampling operator

CONSTANTS DEFINITION CAO, being a cryptographic domain specific language, aims to provide an elegant way to write cryptographic programs. Thus, it is very important to provide a syntax that is very close to standards, as well as a set of types that eases the description of structures used in cryptography. Many cryptographic schemes, specially those who rely on number theory assumptions, contemplate the use of parameters. These parameters are constant numbers that usually depend on each other. Hence, we added the support to define constants in the CAO language. This represents an extension to the CAO syntactic domains.

$$dc := \textbf{def const } x_1 : \tau_1, \ldots, x_n : \tau_n \mid \textbf{def const } x_1 : \tau_1, x_n, \ldots, x_n : \tau_n := e_1, ..., e_n$$

Consequently, a CAO program may now be also composed by constant definitions.

$$
\begin{array}{rcl}
pg & := & \ldots \\
 & & \mid dc \\
 & & \ldots
\end{array}
$$

We also add the ability to define constants inside functions or procedures.

$$
\begin{array}{rcl}
c & := & \ldots \\
 & & \mid dc \\
 & & \ldots
\end{array}
$$

The typechecking rule that deals with constant definitions can be found in Figure 42.

$$\frac{\Delta \vdash_t t \rightsquigarrow \tau}{\Gamma, \Delta \models_\rho \mathsf{def\ const}\ x : t :: (\bullet, \mathsf{Pure}, \Gamma[x :: \tau])} \quad x \notin dom(\Gamma), x \notin dom(\Delta)$$

$$\frac{\Delta \vdash_t t \rightsquigarrow \tau}{\Gamma, \Delta \models_\rho \mathsf{def\ const}\ x : t := e :: (\bullet, \mathsf{Pure}, \Gamma[x :: \tau], \Delta[x := e])} \quad x \notin dom(\Gamma), x \notin dom(\Delta)$$

$$\frac{\Delta \vdash_t t \rightsquigarrow \tau}{\Gamma, \Delta \models_\rho \mathsf{def\ const}\ x_1, \ldots, x_n : t :: (\bullet, \mathsf{Pure}, \Gamma[x_1 :: \tau, \ldots, x_n :: \tau])}$$

where $x_1, \ldots, x_n \notin dom(\Gamma), x_1, \ldots, x_n \notin dom(\Delta), x_i \neq x_j$ for $1 \leq i \leq n$ and $1 \leq j \leq n$

$$\frac{\Delta \vdash_t t \rightsquigarrow \tau}{\Gamma, \Delta \models_\rho \mathsf{def\ const}\ x_1, \ldots, x_n : t := e :: (\bullet, \mathsf{Pure}, \Gamma[x_1 :: \tau, \ldots, x_n :: \tau], \Delta[x_1 := e, \ldots, x_n := e])}$$

where $x_1, \ldots, x_n \notin dom(\Gamma), x_1, \ldots, x_n \notin dom(\Delta), x_i \neq x_j$ for $1 \leq i \leq n$ and $1 \leq j \leq n$

Figure 42: Typechecking rule for constant definitions

REFINEMENTS OF FUNCTION PARAMETERS Since CAO required all sizes to be statically known, implementing algorithms which were designed to handle arguments with variable size was not possible. In order to circumvent this restriction, one had to define multiple versions of the same function, each of them instantiated with a different possible size. This restriction is clearly scalable in most situations, despite reasonable for cryptographic implementations whose parameters are taken from a set of usual values (e.g., size of the key).

In our OCaml implementation of the CAO typechecker, we add the ability of declaring symbolic constants as parameters of functions. Consider the following signature of a function:

```
def f(const n : int { 0 < n }, v : vector[n] of int) : vector[2*n] of int
```

The parameter v is a vector of integers whose size depends on the symbolic constant n, declared as a (positive) symbolic parameter. The return type may also depend on the symbolic parameters of the function; in our example, the size of the return vector must be twice the size of the input vector.

The joint conditions of the symbolic parameters of a function can be seen as its precondition: they are assumed to always hold in the body of the function, and only instantiations that satisfy them are accepted in function calls.

6.3 FORMALISATION OF THE CAO TYPES IN EASYCRYPT

Before applying the translation algorithm between CAO and EasyCrypt, the types of the CAO language need to be correctly formalised in EasyCrypt, so that the there are no lost of semantics during the mapping phase. In order to formalise the CAO types in EasyCrypt, the strategy chosen consisted in developing new EasyCrypt theories - one for each CAO type - and then to clone (instantiate) the corresponding theory with the parameters of the type. For example, supposing that some CAO program uses some variable of type `vector[10] of int`, the *CAO_vector* theory would be cloned with **type** = *CAO_int* and with size = 10, so that elements of the type defined by the instantiated theory refers uniquely to vectors with size 10 and with elements of type *CAO_int*.

In this section, we describe how every type available in the CAO language is translated into an EasyCrypt theory. The theories have the following form. First, there is the definition of the parameters of the type and its constrains - for example, in the case of bit strings, the parameter is the size of the bit string and its condition is that the size must be greater than 0. Next, there is the explicit definition of the logic type that matches the CAO type. Then, the theory specifies all the available operations over that type, as well as an axiomatisation of their behaviour. Finally, a probability distribution is defined for the type.

6.3.1 *Integer type*

The CAO *int* type is directly mapped into the EasyCrypt *int* type (**type** *CAO_int* = *int*). The *Int* theory from EasyCrypt already defines the expected behaviour of a CAO integer. However, there is the need to extend this theory with three new operations/predicates:

- **op** is_prime : *CAO_int* → *bool* - defines the primality test for some integer.
- **op** gcd : *CAO_int* → *CAO_int* → *CAO_int* - greatest common divisor between two integers.
- **op** inverse_mod_n : *CAO_int* → *CAO_int* → *CAO_int* - modular inverse of an integer modulo another integer.

One axiom is provided to describe the interaction between the previous operations: **axiom** prime_gcd (x p : *CAO_int*): is_prime p $\wedge 0 < x < p \Rightarrow$ gcd x p = 1, that states that the greatest common divisor of any number and a prime number is always one.

These operations are kept abstract. Nevertheless, suppose that one is using the type `mod[7]` in his program and that he wants to perform some proof around the division operation. Since 7 is a prime number, the `mod[7]` type defines a field and, therefore, the division operation is defined. One can annotate the program with the precondition `/*@requires is_prime 7 = true*/` in order to discharge all the desired proofs.

6.3.2 *Boolean type*

Similarly to the integer type, the CAO boolean type is directly mapped to the EasyCrypt boolean type - **type** *CAO_bool* = *bool*. There is no need to extend the *Bool* EasyCrypt theory since it defines every operation needed to correctly model the boolean type and the bit string type (that, in our model, is seen as an array of boolean values).

Considering that set of boolean values $\mathbb{B}$ has only two possible values ($true$ or $false$), the boolean sampling probability distribution is easy to define: the probability of $true$ and $false$ is $1/2$. We define this probability using the mu EasyCrypt operation, which gives us the necessary lemmas to reason about boolean distributions.

$$\begin{aligned} \forall p : CAO\_bool \rightarrow CAO\_bool, \quad & \mu\ bool\_distr\ p = 1/2 \\ \forall b : CAO\_bool, \quad & \textbf{in\_supp}\ b\ bool\_distr \\ \forall b : CAO\_bool, \quad & \mu\_x\ bool\_distr\ b = 1/2 \\ & \textbf{is\_lossless}\ bool\_distr \end{aligned}$$

6.3.3 *Ring/field type*

The ring and field CAO type $mod[n]$ corresponds to the $\mathbb{Z}_n$ construction. The formalisation of the *CAO_mod* type begins with the definition of the n parameter: it should be a *CAO_int* and it should always be greater than 0. Otherwise, it is impossible to define any algebraic structure.

We could specify the *CAO_mod* type as a *bounded integer* and reuse what was defined for the *CAO_int* type. However, we defined the new type *CAO_mod* and defined all the operations (addition, subtraction, multiplication, division, modulo and exponentiation) for it, as well as provided an axiomatisation to reason about them. In addition, we also defined the additive inverse and the multiplicative inverse as operations, as well as mapping operations between the *CAO_mod* type and the *CAO_int* type.

Providing that we are building the *CAO_mod* theory from scratch, we fixed two important values of *CAO_mod*, that are used when defining the properties of the elements of the type: the *zero* element (CAO_mod_zero) and the *one* element (CAO_mod_one).

The conversion to integers captures the homomorphism mapping a residue class into the corresponding least residue, whereas the converse operation represents the homomorphism mapping an integer into its residue class. We provide two logic functions

```
op ofint : CAO_int → CAO_mod.
op toint : CAO_mod → CAO_int.
```

and a set of axioms around them

$$
\begin{aligned}
& \mathbf{ofint}\ 0 = CAO\_mod\_zero \\
\forall x : CAO\_int, \quad & 0 \leq x \Rightarrow \mathbf{ofint}\ (x+1) = (\mathbf{ofint}\ n) + CAO\_mod\_one \\
\forall x : CAO\_int, \quad & \mathbf{ofint}\ (-x) = -(\mathbf{ofint}\ x)
\end{aligned}
$$

$$
\begin{aligned}
\forall x : CAO\_mod, \quad & 0 \leq \mathbf{toint}\ x < n \\
\forall x : CAO\_mod, \quad & \mathbf{ofint}\ (\mathbf{toint}\ x) = x \\
\forall x : CAO\_int, \quad & \mathbf{toint}\ (\mathbf{ofint}\ x) = x\%n
\end{aligned}
$$

$$
\begin{aligned}
\forall x : CAO\_int, \quad & x \geq n \Rightarrow \mathbf{toint}\ (\mathbf{ofint}\ x) = \mathbf{toint}\ (\mathbf{ofint}\ (x-n)) \\
\forall x : CAO\_int, \quad & x < 0 \Rightarrow \mathbf{toint}\ (\mathbf{ofint}\ x) = \mathbf{toint}\ (\mathbf{ofint}\ (x+n))
\end{aligned}
$$

The above axioms ensure that the mapping operations operate as expected. The functions ofint and toint are the inverse of one another and, naturally, follow the cancelation rules bellow.

$$
\begin{aligned}
\forall x : CAO\_mod, \quad & \mathbf{ofint}\ (\mathbf{toint}\ x) = x \\
\forall x : CAO\_int, \quad & 0 \leq x < n \Rightarrow \mathbf{toint}\ (\mathbf{ofint}\ x) = x
\end{aligned}
$$

The residue operations are defined abstractly.

```
op ( * ): CAO_mod → CAO_mod → CAO_mod. (* multiplication modulo n *)
op ( + ): CAO_mod → CAO_mod → CAO_mod. (* addition modulo n *)
op [ − ]: CAO_mod → CAO_mod. (* the additive inverse *)
op inv: CAO_mod → CAO_mod. (* the multiplicative inverse *)

op (−) : CAO_mod → CAO_mod → CAO_mod. (* subtraction modulo n *)
op (%) : CAO_mod → CAO_mod → CAO_mod. (* division modulo n for y <> 0 *)
op (%%) : CAO_mod → CAO_mod → CAO_mod. (* modulo modulo n for y <> 0 *)
op (^) : CAO_mod → CAO_int → CAO_mod. (* exponentiation *)
```

Again, we could make use of the ofint and toint functions to rely on the EasyCrypt *Int* theory and explicitly define the arithmetic operations. For example, the addition operation could be written as **op** (+)(x y : *CAO_mod*): *CAO_mod* = ofint (toint x + toint y). Note that, considering the previously defined axioms, the two definitions for the operations are equivalent and this statement can be proved using a simple SMT call. We provide an example for the addition operation.

$$\forall x, y : CAO\_mod, x + y = \textbf{ofint}\ (\textbf{toint}\ x + \textbf{toint}\ y)$$

The properties of the ring or field operations are also axiomatised. If $\mathbb{Z}_n$ defines a ring, then the following properties will hold for the addition and multiplication operations:

- Commutativity of the addition - $\forall x, y : CAO\_mod,\ x + y = y + x$
- Associativity of the addition - $\forall x, y, z : CAO\_mod,\ x + (y + z) = (x + y) + z$
- Identity element of the addition - $\forall x : CAO\_mod,\ x + CAO\_mod\_zero = x$
- Additive inverse - $\forall x : CAO\_mod,\ x + -x = CAO\_mod\_zero$
- Associativity of the multiplication - $\forall x, y, z : CAO\_mod,\ x * (y * z) = (x * y) * z$

From the properties above, we are able to define the subtraction $x - y$ as the addition of x with the additive inverse of y: **axiom** sub_def (x y:*CAO_mod*): x y = x + y.. If $\mathbb{Z}_n$ defines a field, it will inherit all the properties of the ring algebraic structure with some additional ones:

- Commutativity of the multiplication - $\forall x, y : CAO\_mod,\ \textbf{is\_prime}\ n \Rightarrow x * y = y * x$
- Identity element of the multiplication - $\forall x : CAO\_mod,\ \textbf{is\_prime}\ n \Rightarrow x * CAO\_mod\_one = x$
- Multiplicative inverse - $\forall x : CAO\_mod,\ \textbf{is\_prime}\ n \Rightarrow x \neq CAO\_mod\_zero \Rightarrow (x * (invx)) = CAO\_mod\_one$
- Distributivity of the multiplication over the addition - $\forall x, y, z : CAO\_mod,\ \textbf{is\_prime}\ n \Rightarrow x * y + x * z = x * (y + z)$

Notice that we add the condition is_prime n to restrict the property to instantiations of the *CAO_mod* theory that define fields.

The modeling of the operations concludes with the definition of the division operation. The division is always defined when $\mathbb{Z}_n$ is a field but is also defined for some cases when $\mathbb{Z}_n$ is a ring. More concretely, the division operation is defined if the denominator is co-prime with n, i.e., if the greatest common divisor between the denominator and n is 1. The division is, therefore, defined by the axiom **axiom** div_def (x y:*CAO_mod*): gcd n (toint y)= 1 $\Rightarrow$ x /% y = x $*$ (inv y).

Our model also contemplates two axioms to reason about the greatest common divisor operation combined with the multiplicative inverse of some number.

$$\forall x : CAO\_mod, \quad \mathbf{gcd}\ (\mathbf{toint}\ x)\ n = 1 \Rightarrow \mathbf{toint}\ x * \mathbf{toint}\ (\mathbf{inv}\ x) = 1$$
$$\forall x : CAO\_mod, \quad y : CAO\_int, \mathbf{toint}\ x * y = 1 \Rightarrow \mathbf{toint}\ (\mathbf{inv}\ x) = y$$

The probability distribution mod_distr over the *CAO_mod* type is defined in the obvious way. Seeing the elements of $\mathbb{Z}_n$ as the set $[0, n[$, with n elements, the probability of an element of this set being sampled is $1/n$, in constrast to the probability of sampling an element outside of the set, which is 0 - is not in the support of the distribution. This distribution is, naturally, *lossless* and uniform. The following lemmas model mod_distr.

$$\forall x : CAO\_mod, \quad 0 \leq \mathbf{toint}\ x < n \Rightarrow \mu\_x\ mod\_distr\ x = 1/n$$
$$\forall x : CAO\_mod, \quad !(0 \leq \mathbf{toint}\ x < n) \Rightarrow \mu\_x\ mod\_distr\ x = 0$$
$$\mathbf{is\_lossless}\ mod\_distr$$
$$\mathbf{is\_uniform}\ mod\_distr$$

The *CAO_mod* theory finishes with a set of lemmas that we were able to prove about the *CAO_mod* type. These lemmas do not correspond to properties of the ring or field algebraic structures but can be important auxiliary lemmas when performing more complex proofs.

$$\forall x : CAO\_mod, \quad 0 \leq \mathbf{toint}\ x < n$$
$$\forall x : CAO\_int, \quad 0 \leq x < n \Rightarrow \mathbf{toint}\ (\mathbf{ofint}\ x) = x$$
$$\forall x : CAO\_mod, \quad (x - x) = CAO\_mod\_zero$$
$$\forall x : CAO\_mod, \quad CAO\_mod\_zero + x = x$$
$$\forall x : CAO\_mod, \quad \mathbf{is\_prime}\ n \Rightarrow x * CAO\_mod\_zero = CAO\_mod\_zero$$
$$\forall x, y : CAO\_mod, \quad (-x) * y = -(x * y)$$
$$\forall x, y : CAO\_mod, \quad y * (-x) = -(y * x)$$
$$\forall x : CAO\_mod, \quad -(-x) = x$$
$$\forall x, y, z : CAO\_mod, \quad x * y - x * z = x * (y - z)$$
$$\forall x : CAO\_mod, \quad \mathbf{is\_prime}\ n \Rightarrow (-CAO\_mod\_one) * x = -x$$
$$\forall x, y : CAO\_mod, \quad (-x) + (-y) = -(x + y)$$
$$\forall x : CAO\_mod, \quad 0 \leq \mathbf{toint}\ x$$
$$\forall x : CAO\_mod, \quad \mathbf{toint}\ x < n$$
$$\forall x : CAO\_mod, \quad \mathbf{toint}\ x \leq n - 1$$
$$\forall x : CAO\_int, \quad 0 \leq x < n \Rightarrow \mathbf{toint}\ (\mathbf{ofint}\ x) = x$$
$$\mathbf{ofint}\ 1 = CAO\_mod\_one$$

6.3.4 *Register int type*

The register int CAO type corresponds to machine bounded integers. The semantics of register integers is similar to the semantics of $mod[2^w]$, with w defined by the machine. Therefore, we could define a CAO register int using the *CAO_mod* theory. However, this representation would create a major overhead in the translated code, since we would require a lot of unecessary conversions between

the *CAO_int* and the *CAO_mod* type in, for example, loop control variables. Thus, we define a new theory, *CAO_reg_int* that describes the behaviour of the CAO register int type.

Similarly to the *CAO_mod* theory, the *CAO_reg_int* theory starts by defining of a constant parameter w that will be used to represent the bound of a register int. The *CAO_reg_int* type will then be specified as being equal to the *CAO_int*. Next, we define the possible values of the inhabitants of the *CAO_reg_int* type by the following axiom

$$\forall x : CAO\_reg\_int,\ 0 \Leftarrow x < 2^w$$

We also provide the characterisation of an uniform probability distribution over the *CAO_reg_int* type. The representation of this probability distribution is equal to the *CAO_mod* probability distribution. The only difference is that, instead of considering the elements of the type are bounded by the interval $[0, n[$, they are bounded by the interval $[0, 2^w[$

6.3.5 *Bit string type*

A bit string in CAO is parametrised by its size and one is not able to define a bit string of arbitrary length. Thus, the definition of the theory for the bit string type starts by define the size of the bit string as a constant of type *CAO_int* and by restricting its size to values greater than 0.

```
const size : CAO_int.
axiom size_pos : CAO_bits.size → 0.
```

The CAO bit string type is directly mapped into the EasyCrypt *bitstring* type.

```
type CAO_bits = bitstring.
```

We define two initial values for *CAO_bits*: a bit string of elements 1 and a bit string of elements 0. We rely on the zeros and ones operations already defined in the *Bitstring* EasyCrypt theory. These operations generate *zero* bit strings and *one* bit strings, respectively.

```
op CAO_bits_zero = zeros size.
op CAO_bits_one = ones size.
```

To model the available bit string operations we add two auxiliary logic functions to our model: shift and _blit. Informally, the first takes as input a bit string, starting in position 0, and produces the bit string that starts at position i. The second involves two vectors, source s and destination d, an index i and a length parameter l. It produces the vector with the contents of d for indices 0 to $i-1$, and from $i+l$ onwards; the l positions in between contain the region $0..l-1$ of s. We rely on the blit

EasyCrypt function to the define our _blit function. The behaviour of these functions is modeled by the axioms given bellow.

$$\begin{aligned} \forall v : CAO\_bits,\ ofs, i : CAO\_int, \quad & 0 \leq i < size \Rightarrow (\mathbf{shift}\ v\ ofs).[i] = v.[ofs + i] \\ \forall src, dst : CAO\_bits, ofs, len : CAO\_int), \quad & 0 \leq ofs \Rightarrow 0 \leq len \Rightarrow ofs + len \leq \mathbf{length}\ dst \Rightarrow \\ & ofs + len \leq \mathbf{length}\ src \Rightarrow \\ & \mathbf{\_blit}\ src\ dst\ ofs\ len = \mathbf{blit}\ dst\ ofs\ src\ 0\ len \end{aligned}$$

We do not provide a concatenation operation in our model. Instead, we rely on the concatenation operation || that is already defined and axiomatised in EasyCrypt for container types. The bit string operations are modeled as follows. Note that the range selection is defined with recursion to the sub EasyCrypt function.

$$\begin{aligned} \forall v : CAO\_bits,\ i, j : CAO\_int, \quad & \mathbf{rsel}\ v\ i\ j = \mathbf{sub}\ v\ i\ j \\ \forall src, dst : CAO\_bits,\ i, j : CAO\_int, \quad & \mathbf{rass}\ src\ i\ j\ dst = \mathbf{\_blit}\ dst\ src\ i\ (j - i + 1) \\ \forall v : CAO\_bits,\ i : CAO\_int, \quad & v < |i = ((\mathbf{rsel}\ v\ (\mathbf{length}\ v - i)\ (\mathbf{length}\ v - 1))\ || \\ & (\mathbf{rsel}\ v\ 0\ (\mathbf{length}\ v - i - 1))) \\ \forall v : CAO\_bits,\ i : CAO\_int, \quad & v| > i = ((\mathbf{rsel}\ v\ i(\mathbf{length}\ v - 1))\ ||\ (\mathbf{rsel}\ v\ 0(i - 1))) \\ \forall v : CAO\_bits,\ i : CAO\_int, \quad & v << i = (\mathbf{\_blit}\ (\mathbf{shift}\ v\ 0)\ (\mathbf{zeros}\ (\mathbf{length}\ v))\ i\ ((\mathbf{length}\ v) - i)) \\ \forall v : CAO\_bits,\ i : CAO\_int, \quad & v >> i = (\mathbf{\_blit}\ (\mathbf{zeros}\ (\mathbf{length}\ v))\ (\mathbf{shift}\ v\ 0)\ ((\mathbf{length}\ v) - i)i) \end{aligned}$$

The bitwise operations are defined explicitly in our model. We define them with recursion to the EasyCrypt map2 function. Intuitively, the map2 is similar to the map function, except that it takes two container types and performs the mapping of the given function element-wise. The negation of a bit string (negation of all the elements of the bit string) is defined using the simple map operation. The following piece of EasyCrypt code defines all the available bitwise operations.

```
op ( ^^ ) (bs0 bs1: CAO_bits): CAO_bits = map2 (Bool.(^^)) bs0 bs1. (* Bitwise XOR *)
op ( && ) (bs0 bs1: CAO_bits): CAO_bits = map2 (∧) bs0 bs1. (* Bitwise AND *)
op ( || ) (bs0 bs1: CAO_bits): CAO_bits = map2 (∨) bs0 bs1. (* Bitwise OR *)
op [!] (bs : CAO_bits): CAO_bits = map ([!]) bs. (* Negation of a bitstring *)
```

To finish the formalisation of the operations over bit strings, we define the cast to/for the integer type and we add the corresponding cancelation axioms.

```
op bits_of_int : CAO_bits → CAO_int.
op int_of_bits : CAO_int → CAO_bits.
```

$$\begin{aligned} \forall x : CAO\_int, \quad & \mathbf{bits\_of\_int}\ (\mathbf{int\_of\_bits}\ x) = x \\ \forall x : CAO\_bits, \quad & \mathbf{int\_of\_bits}\ (\mathbf{bits\_of\_int}\ x) = x \end{aligned}$$

The probability distribution bits_distr is established with two axioms: when the length of the bit string corresponds to the size parameter, the probability of sampling some bit string is $1/(2^{size})$, whereas when the length does not correspond to the size parameter there is no probability of sampling any bit string. The definition of the distribution is concluded with the lemmas that reason about the support of the distribution and about its weight.

$$
\begin{aligned}
&\forall s : CAO\_bits, && \mathbf{length}\ s = size \Rightarrow \mu\_x\ bits\_distr\ s = 1/(2^{size}) \\
&\forall s : CAO\_bits, && \mathbf{length}\ s <> size \Rightarrow \mu\_x\ bits\_distr\ s = 0 \\
&\forall s : CAO\_bits, && \mathbf{in\_supp}\ s\ bits\_distr \iff \mathbf{length}\ s = size \\
& && 0 \leq size \Rightarrow \mathbf{is\_lossless}\ bits\_distr \\
& && size < 0 \Rightarrow !(\mathbf{is\_lossless}\ bits\_distr)
\end{aligned}
$$

We conclude the *CAO_bits* theory with the inclusion of some lemmas that were taken from the previous CAOVerif formalisation and we prove that they hold in our model.

$$
\begin{aligned}
&\forall v1, v2 : CAO\_bits, && v1 == v2 \iff \mathbf{length}\ v1 = \mathbf{length}\ v2 \\
& && \wedge \forall i : CAO\_int, 0 \leq i < \mathbf{length}\ v1 \Rightarrow v1.[i] = v2.[i] \\
&\forall len, i : CAO\_int, v : CAO\_bits, && v == \mathbf{zeros}\ len \Rightarrow 0 \leq i < len \Rightarrow v.[i] = false \\
&\forall v1, v2 : CAO\_bits, i : CAO\_int, && v1 == v2 \Rightarrow v1.[i] = v2.[i] \\
&\forall v : CAO\_bits, i : CAO\_int, x : CAO\_bool, && 0 \leq i < \mathbf{length}\ v \Rightarrow (v.[i \leftarrow x]).[i] = x \\
&\forall v : CAO\_bits, i, j : CAO\_int, x : CAO\_bool, && 0 \leq i < \mathbf{length}\ v \Rightarrow 0 \leq j < \mathbf{length}\ v \Rightarrow i \neq j \Rightarrow \\
& && (v.[i \leftarrow x]).[j] = v.[j] \\
&\forall src, dst : CAO\_bits, ofs, len, i : CAO\_int, && 0 \leq ofs \Rightarrow 0 \leq len \Rightarrow ofs + len \leq \mathbf{length}\ dst \Rightarrow \\
& && ofs + len \leq \mathbf{length}\ src \Rightarrow i \geq ofs \wedge i < ofs + len \Rightarrow \\
& && (\_\mathbf{blit}\ src\ dst\ ofs\ len).[i] = src.[i - ofs] \\
&\forall src, dst : CAO\_bits, ofs, len, i : CAO\_int, && 0 \leq ofs \Rightarrow 0 \leq len \Rightarrow ofs + len \leq \mathbf{length}\ dst \Rightarrow \\
& && ofs + len \leq \mathbf{length}\ src \Rightarrow i < ofs \leq i \leq ofs + len \Rightarrow \\
& && (\_\mathbf{blit}\ src\ dst\ ofs\ len).[i] = dst.[i] \\
&\forall b1, b2 : CAO\_bool, && b1 = b2 \Rightarrow b1\hat{}b2 = false \\
&\forall b1, b2 : CAO\_bool, && b1 \neq b2 \Rightarrow b1\hat{}b2 = true \\
&\forall v1, v2 : CAO\_bits, i : CAO\_int, && \mathbf{length}\ v1 = \mathbf{length}\ v2 \Rightarrow 0 \leq i < \mathbf{length}\ v1 \Rightarrow \\
& && (v1\hat{}v2).[i] = v1.[i]\hat{}v2.[i] \\
&\forall v1, v2 : CAO\_bits, i : CAO\_int, && \mathbf{length}\ v1 = \mathbf{length}\ v2 \Rightarrow 0 \leq i < \mathbf{length}\ v1 \Rightarrow \\
& && (v1\ ||\ v2).[i] = v1.[i] \vee v2.[i] \\
&\forall v1, v2 : CAO\_bits, i : CAO\_int, && \mathbf{length}\ v1 = \mathbf{length}\ v2 \Rightarrow 0 \leq i < \mathbf{length}\ v1 \Rightarrow \\
& && (v1\ \&\&\ v2).[i] = v1.[i] \wedge v2.[i] \\
&\forall b : CAO\_bits, i : CAO\_int, && 0 \leq i < \mathbf{length}\ b \Rightarrow (!b).[i] = !b.[i] \\
&\forall v1, v2 : CAO\_bits, && \mathbf{length}\ (v1||v2) = \mathbf{length}\ v1 + \mathbf{length}\ v2
\end{aligned}
$$

6.3.6 *Extension field type*

Consider the following CAO type declarations

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

Take the field extension type $GF2N$. Types of this form are also algebraic types that model the Galois field (finite field) of order n^d, where n is a prime number and d is the degree of the irreducible polynomial $p(X)$. We emphasize that, in CAO, each such type represents a specific construction of an extension field, whose representation is fixed as elements of the polynomial ring $Z_n[X]$, and the semantics of operations is defined based on polynomial arithmetics modulo $p(X)$. Furthermore this type is only valid when n is prime and $p(X)$ is irreducible.

We rely on the $Poly$ EasyCrypt theory to define the extension field type. We start by defining the type of the coefficients of the polynomials - which can be elements of type Z_n or elements of type $Z_n[X]$ - and by defining the *zero* element of the type of the coefficients (important to define properties over the exponentiation operation). After, the $Poly$ theory is cloned giving the type of the coefficients.

```
clone Poly as Poly_CAO_mod_q with
  type R = coef_type.
```

This cloning gives us a new type: the polynomials with coefficients of the type coef_type and a set of axioms and lemmas to reason about it. We define the type *CAO_exfield* to be equal to the type that results from the cloning and provide the *zero* and *one* elements of the *CAO_exfield*.

We define the *set* and *get* operations to reason about *CAO_exfield* coefficients, casts to and from list and a function to get the degree of a polynomial. The last four functions are defined with recursion to EasyCrypt functions.

```
op "_[_]" (p : CAO_exfield, i : CAO_int) = Poly_CAO_mod_q.coeff p i.
op "_[_<-_]" : CAO_exfield → CAO_int → coef_type → CAO_exfield.
op tolist = Poly_CAO_mod_q.toseq.
op fromlist = Poly_CAO_mod_q.ofseq.
op deg = Poly_CAO_mod_q.deg.
```

The *get* and *set* operations are defined by the following lemmas.

$$\begin{aligned}
\forall p : CAO_exfield, i : CAO_int, x : coef_type, \quad & p.[i \leftarrow x] = \mathbf{fromlist}\ ((\mathbf{tolist} p).[i \leftarrow x]) \\
\forall p : CAO_exfield, i, j : CAO_int, x : coef_type, \quad & 0 \leq i < \mathbf{deg} p \Rightarrow 0 \leq j < \mathbf{deg} p \Rightarrow i \neq j \Rightarrow \\
& (p.[i \leftarrow x]).[j] = p.[j] \\
\forall i : CAO_int, \quad & 0 \leq i < \mathbf{deg} CAO_exfield_zero \Rightarrow \\
& CAO_exfield_zero.[i] = coef_type_zero
\end{aligned}$$

The arithmetic operations available in the *CAO_exfield* type are the same as in the *CAO_mod* and the field properties are defined in the same way. The only detail relevant to mention is that we match the addition and the unary minus operations with the ones already defined in the polynomial theory.

```
op ( + ) = Poly_CAO_mod_q.ZModule.(+).
op [ − ] = Poly_CAO_mod_q.ZModule.([−]).
```

There are two available casts in the *CAO_exfield* type. One can map an element of the *CAO_exfield* type to an element of the type of the coefficients (and vice-versa) and also one is able to map elements of the *CAO_exfield* type to vectors, resulting on a vector made of the polynomial coefficients.

```
op ofmod : coef_type → CAO_exfield.
op tomod : CAO_exfield → coef_type.

op ofexfield : CAO_exfield → CAO_vector.
op toexfield : CAO_vector → CAO_exfield.
```

The definition of the probability distribution over the type *CAO_exfield* is kept abstract, i.e., no probability is defined for the sampling operation. However, we provide a set of axioms that state that the distribution is lossless, uniform and that the elements of the type *CAO_exfield* are in the support of the distribution.

$$\begin{aligned} & \textbf{is\_lossless}\ exfield\_distr \\ & \textbf{is\_uniform}\ exfield\_distr \\ \forall x : CAO\_exfield, \quad & \textbf{in\_supp}\ x\ exfield\_distr \end{aligned}$$

Again, we end the concretisation of the *CAO_exfield* theory with a set of proved lemmas about the *CAO_exfield* type and its operations.

$$\begin{aligned} \forall p1, p2 : CAO\_exfield, i : CAO\_int, \quad & \textbf{deg}\ p1 = \textbf{deg}\ p2 \Rightarrow 0 \leq i < \textbf{deg}\ p1 \Rightarrow p1 = p2 \Rightarrow \\ & p1.[i] = p2.[i] \\ \forall p1, p2 : CAO\_exfield, \quad & p1 \neq CAO\_exfield\_zero \Rightarrow p2 \neq CAO\_exfield\_zero \Rightarrow \\ & p1 * p2 \neq CAO\_exfield\_zero \\ \forall p1, p2 : CAO\_exfield, \quad & p1 \neq CAO\_exfield\_zero \Rightarrow p2 \neq CAO\_exfield\_zero \Rightarrow \\ & p1/p2 \neq CAO\_exfield\_zero \\ \forall p1, p2 : CAO\_exfield, \quad & p1 \neq p2 \Rightarrow p1 - p2 \neq CAO\_exfield\_zero \\ \forall p1, p2 : CAO\_exfield, \quad & p1 \neq -p2 \Rightarrow p1 + p2 \neq CAO\_exfield\_zero \\ \forall p : CAO\_exfield, \quad & p \neq CAO\_exfield\_zero \Rightarrow -p \neq CAO\_exfield\_zero \end{aligned}$$

6.3.7 *Vector type*

The vector type *CAO_vector* is parametrised by an integer *size*, that represents the size of the vector, and by a type t, representing the type of the elements of the container. With both parameters, we uniquely define the vector of type t with constant size *size*. Our *CAO_vector* theory makes use of the EasyCrypt Array theory - we realise the CAO vectors as arrays of finite length. By relying on the EasyCrypt Array theory, we inherit a lot of operations and axioms, thus reducing the overhead and the possible mistakes when defining our theory. For example, stating that a *CAO_vector* type is equal to the *array* type, we no longer need to define the *get* and *set* operations and we are able to enrich the *CAO_vector* type with, for example, the *map* operations. The type is, then, defined as

```
type CAO_vector = t array.
```

We define the length of the elements of the *CAO_vector* type as a constant function CAO_vector_length , that always output the value of *size*.

$$\forall x : CAO\_vector, \mathbf{CAO\_vector\_length}\ x = size$$

Additionally, we provide a function to initialise vectors with a given element. This function only takes as input the element with which the vector will be initialised, since the length is already defined to be *size*. This operation CAO_vector_make is modeled as follows:

$$\begin{aligned} \forall x : t, &\quad \mathbf{CAO\_vector\_length}\ (\mathbf{CAO\_vector\_make}\ x) = size \\ \forall x : t, i : CAO\_int, &\quad 0 \leq i < size \Rightarrow (\mathbf{CAO\_vector\_make}\ x).[i] = x \end{aligned}$$

The available operations over the *CAO_vector* are the same as the available for the *CAO_bits* type (Section 6.3.5) - except for the bitwise operations - and are specified and axiomatised the same way. The only exception is the shift ($<<$ and $>>$) operations. The shift operation implies that the values of a vector are moved right or left and that the remaining values are filled with the *zero* value. Therefore, we need to know the exact type of the elements of the vector to defined the shift operations using the *zero* value of the type of the elements. We provide an example of the definition of the shift operation for vectors with elements of the *CAO_mod* type.

```
op ( >> ) : α array → CAO_int → α array.
op ( << ) : α array → CAO_int → α array.
```

$$\forall v : CAO\_mod\ array, i : CAO\_int, \quad v << i = (\_\mathbf{blit}\ (\mathbf{shift}\ v\ 0)\ (\mathbf{make}\ (\mathbf{length}\ v)\ CAO\_mod\_zero)\ i\ ((\mathbf{length}\ v) - i))$$

$$\forall v : CAO\_mod\ array, i : CAO\_int, \quad v >> i = (\_\mathbf{blit}\ (\mathbf{make}\ (\mathbf{length}\ v)\ CAO\_mod\_zero)\ (\mathbf{shift}\ v\ 0)\ ((\mathbf{length}\ v) - i) i)$$

Note that we define the operations using the type α *array* (an array of some type α) instead of using *CAO_vector*. This way, we can reason about arrays of different types and, since the *CAO_vector* is an array of a specific type, it inherits all these operations and its properties.

A probability distribution over a vector vector_distr is naturally influenced by the probability distribution over the type of the elements of the vector. We define the probability of the sampling from vector_distr to output a given vector as follows

$$\forall d : t\ distr, x : CAO\_vector, \quad size = \mathbf{CAO\_vector\_length}\ x \Rightarrow \mu\_x\ vector\_distr\ x = \mathbf{fold\_right}\ (fun\ p\ x, p * \mu\_x\ d\ x)\ 1\ x$$

Informally, the above lemma says that the probability of vector_distr outputing a specific vector is the probability of outputing a specific element of the type of the vector in every slot of the vector. Notice that we only define the distribution for *CAO_vector* elements of the correct length.

The concretisation of the probability distribution over the *CAO_vector* type is completed with the definition of the following axioms and lemmas to reason about the support of the distribution, its weight and its uniform property.

$$\forall x : CAO\_vector, d : t\ distr, \quad 0 \leq size \Rightarrow \mathbf{in\_supp}\ x\ vector\_distr \iff (\mathbf{CAO\_vector\_length}\ x = size \wedge \mathbf{all}\ (\mathbf{support}\ d)\ x)$$

$$\forall d : t\ distr, x : CAO\_vector, \quad 0 \leq size \Rightarrow (\forall y, \mathbf{in\_supp}\ y\ d) \Rightarrow \mathbf{length}\ x = size \iff \mathbf{in\_supp}\ x\ vector\_distr$$

$$\forall x : CAO\_vector, d : tdistr, \quad 0 \leq size \Rightarrow \mathbf{in\_supp}\ x\ vector\_distr \Rightarrow \mathbf{length}\ x = size$$

$$\forall x : CAO\_vector, d : tdistr, k : CAO\_int, \quad 0 \leq k < size \Rightarrow \mathbf{in\_supp}\ x\ vector\_distr \Rightarrow \mathbf{in\_supp}\ x.[k]\ d$$

$$\forall d : t\ distr, \quad 0 \leq size \Rightarrow \mathbf{weight}\ vector\_distr = (\mathbf{weight}\ d)\hat{}size$$

$$\forall d : t\ distr, \quad 0 \leq size \Rightarrow \mathbf{is\_lossless}\ d \Rightarrow \mathbf{is\_lossless}\ vector\_distr$$

$$\forall d : t\ ; distr, \quad \mathbf{is\_uniform}\ d \Rightarrow \mathbf{is\_uniform}\ vector\_distr$$

To end the *CAO_vector* theory, we proved that the following lemmas, taken from the original CAOVerif formalisation, still hold in our model.

$$
\begin{aligned}
&\forall v1, v2 : CAO\_vector, && v1 = v2 \iff \textbf{CAO\_vector\_length}\ v1 = \\
&&& \textbf{CAO\_vector\_length}\ v2 \wedge \forall i : CAO\_int, \\
&&& 0 \leq i < \textbf{CAO\_vector\_length}\ v1 \Rightarrow v1.[i] = v2.[i] \\
&\forall v : CAO\_vector,\ i : CAO\_int,\ x : t, && 0 \leq i < \textbf{CAO\_vector\_length}\ v \Rightarrow (v.[i \leftarrow x]).[i] = x \\
&\forall v : CAO\_vector,\ i, j : CAO\_int,\ x : t, && 0 \leq i < \textbf{CAO\_vector\_length}\ v \Rightarrow \\
&&& 0 \leq j < \textbf{CAO\_vector\_length}\ v \Rightarrow i \neq j \Rightarrow \\
&&& (v.[i \leftarrow x]).[j] = v.[j] \\
&\forall src, dst : CAO\_vector,\ ofs, len, i : CAO\_int, && 0 \leq ofs \Rightarrow 0 \leq len \Rightarrow \\
&&& ofs + len \leq \textbf{CAO\_vector\_length}\ dst \Rightarrow \\
&&& ofs + len \leq \textbf{CAO\_vector\_length}\ src \Rightarrow \\
&&& i \geq ofs \wedge i < ofs + len \Rightarrow \\
&&& (\textbf{\_blit}\ src\ dst\ ofs\ len).[i] = src.[i - ofs] \\
&\forall src, dst : CAO\_vector,\ ofs, len, i : CAO\_int, && 0 \leq ofs \Rightarrow 0 \leq len \Rightarrow \\
&&& ofs + len \leq \textbf{CAO\_vector\_length}\ dst \Rightarrow \\
&&& ofs + len \leq \textbf{CAO\_vector\_length}\ src \Rightarrow \\
&&& i < ofs \wedge i \geq ofs + len \Rightarrow \\
&&& (\textbf{\_blit}\ src\ dst\ ofs\ len).[i] = dst.[i] \\
&\forall v1, v2 : CAO\_vector, && \textbf{CAO\_vector\_length}\ (v1 \mid\mid v2) = \\
&&& \textbf{CAO\_vector\_length}\ v1 + \textbf{CAO\_vector\_length}\ v2
\end{aligned}
$$

6.3.8 *Matrix type*

Matrices in CAO are parametrised by three elements: the number of rows (n_rows) and the number of columns (n_columns) of the matrix and the type (t) of the elements that compose it. In order to define the *CAO_matrix* type and theory, we reutilise what is already defined in the Matrix theory of EasyCrypt. Simlar to what was done to the *CAO_vector* theory, the CAO matrix type is defined as follows

```
type CAO_matrix = x matrix.
```

Our *CAO_matrix* theory contemplates the same operations available for the *CAO_vector* theory. We do not provide the definitions for these functions since they are the similar in both theory - with the difference of the operations being defined for two dimensions in the *CAO_matrix* theory. We still provide the definitions for the *make* and *size* operations.

```
op CAO_matrix_size: CAO_matrix → CAO_int * CAO_int.

op CAO_matrix_make: t → CAO_matrix.
```

Since the matrix type is an algebraic type in CAO, we extend our the matrix theory with algebraic operations: addition, subtraction, unary subtraction, multiplication and exponentiation. We follow the same approach used to specify operations for vectors: we define them for the EasyCrypt type to be able to reason about the type of the elements of the container type and then the operations are inherit by our *CAO_matrix* type.

```
op ( + ) : α matrix → α matrix → α matrix.
op ( − ) : α matrix → α matrix → α matrix.
op [ − ] : α matrix → α matrix.
op ( ∗ ) : α matrix → α matrix → α matrix.
op ( ^ ) : α matrix → CAO_int → α matrix.
```

The sizes of the matrix that result from these operations are modelled using the following axioms.

$$
\begin{aligned}
\forall m1, m2 :' a\ matrix, \quad & \mathbf{CAO\_matrix\_size}\ m1 = \mathbf{CAO\_matrix\_size}\ m2 \Rightarrow \\
& \mathbf{CAO\_matrix\_size}\ (m1 + m2) = \mathbf{CAO\_matrix\_size}\ m1 \\
\forall m1, m2 :' a\ matrix, \quad & \mathbf{CAO\_matrix\_size}\ m1 = \mathbf{CAO\_matrix\_size}\ m2 \Rightarrow \\
& \mathbf{CAO\_matrix\_size}\ (m1 - m2) = \mathbf{CAO\_matrix\_size}\ m1 \\
\forall m :' a\ matrix, \quad & \mathbf{CAO\_matrix\_size}\ (-m) = \mathbf{CAO\_matrix\_size}\ m \\
\forall m :' a\ matrix, x : CAO\_int, \quad & \mathbf{CAO\_matrix\_size}\ (m^x) = \mathbf{CAO\_matrix\_size}\ m \\
\forall m1, m2 :' a\ matrix, \quad & \mathbf{snd}\ (\mathbf{CAO\_matrix\_size}\ m1) = \mathbf{fst}\ (\mathbf{CAO\_matrix\_size}\ m2) \Rightarrow \\
& \mathbf{fst}\ (\mathbf{CAO\_matrix\_size}\ (m1\ *\ m2)) = \mathbf{fst}\ (\mathbf{CAO\_matrix\_size}\ m1) \wedge \\
& \mathbf{snd}\ (\mathbf{CAO\_matrix\_size}(m1\ *\ m2)) = \mathbf{snd}\ (\mathbf{CAO\_matrix\_size}\ m2)
\end{aligned}
$$

We use matrix with elements of the *CAO_mod* type to exemplify the axiomatisation of the arithmetic operations over matrices. Note that we do not provide an explicit definition of matrix multiplication. In order to do so, we would need to provide a pointwise product operation between two vectors, however, in CAO, vectors are simply containers and are not considered algebraic types, which invalidates the definition of any algebraic operation around them.

$$\begin{aligned}
&\forall m1, m2 : CAO_mod\ matrix, i, j : CAO_int, && \mathbf{CAO_matrix_size}\ m1 = \mathbf{CAO_matrix_size}\ m2 \Rightarrow \\
&&& 0 \leq i < \mathbf{fst}\ (\mathbf{CAO_matrix_size}\ m1) \Rightarrow \\
&&& 0 \leq j < \mathbf{snd}\ (\mathbf{CAO_matrix_size}\ m1) \Rightarrow \\
&&& (m1 + m2).[(i,j)] = m1.[(i,j)] + m2.[(i,j)] \\
&\forall m1, m2 : CAO_mod\ matrix, i, j : CAO_int, && \mathbf{CAO_matrix_size}\ m1 = \mathbf{CAO_matrix_size}\ m2 \Rightarrow \\
&&& 0 \leq i < \mathbf{fst}\ (\mathbf{CAO_matrix_size}\ m1) \Rightarrow \\
&&& 0 \leq j < \mathbf{snd}\ (\mathbf{CAO_matrix_size}\ m1) \Rightarrow \\
&&& (m1 - m2).[(i,j)] = m1.[(i,j)] - m2.[(i,j)] \\
&\forall m : CAO_mod\ matrix, i, j : CAO_int, && 0 \leq i < \mathbf{fst}\ (\mathbf{CAO_matrix_size}\ m) \Rightarrow \\
&&& 0 \leq j < \mathbf{snd}\ (\mathbf{CAO_matrix_size}\ m) \Rightarrow \\
&&& (-m).[(i,j)] = -m.[(i,j)] \\
&\forall m : CAO_mod\ matrix, i, j, e : CAO_int, && 0 \leq i < \mathbf{fst}\ (\mathbf{CAO_matrix_size}\ m) \Rightarrow \\
&&& 0 \leq j < \mathbf{snd}\ (\mathbf{CAO_matrix_size}\ m) \Rightarrow \\
&&& (m^e).[(i,j)] = m.[(i,j)]^e
\end{aligned}$$

In order to define the probability distribution over matrices, we start by extending the EasyCrypt matrix theory with a *fold_right* function and an *all* predicate, that checks if some predicate holds for all elements of a matrix. These functions are define as follows.

```
op fold_right: ('state → χ → 'state) → 'state → χ matrix → 'state.

op all: (χ → bool) → χ matrix → bool.
```

$$\begin{aligned}
&\forall xss : CAO_matrix, f :' state \rightarrow' x \rightarrow' state, s : t, && \mathbf{CAO_matrix_size}\ size\ xss = (0,0) \Rightarrow \\
&&& (\mathbf{fold_right}\ f\ s\ xss) = s \\
&\forall f :' state \rightarrow' x \rightarrow' state, s : t, xss : CAO_matrix, && 0 < \mathbf{fst}\ (\mathbf{CAO_matrix_size}\ xss) \Rightarrow \\
&&& 0 < \mathbf{snd}(\mathbf{CAO_matrix_size}\ xss) \Rightarrow \\
&&& (\mathbf{fold_right}\ f\ s\ xss) = \mathbf{fold_right}\ f\ (f\ s\ xss.[(0,0)]) \\
&&& (\mathbf{sub}\ xss\ (1,1)\ ((\mathbf{fst}\ (\mathbf{CAO_matrix_size}\ xss) - 1), \\
&&& (\mathbf{snd}\ (\mathbf{CAO_matrix_size}\ xss) - 1))) \\
&\forall p : t \rightarrow CAO_bool, xss : CAO_matrix, && \mathbf{all}\ p\ xss \iff \\
&&& (\forall i, j : CAO_int, 0 \leq i < \mathbf{fst}\ (\mathbf{CAO_matrix_size}\ xss) \Rightarrow \\
&&& 0 \leq j < \mathbf{snd}(\mathbf{CAO_matrix_size}\ xss) \Rightarrow p\ xss.[(i,j)])
\end{aligned}$$

The definition of the probability distribution over matrices is similar to the definition of the probability distribution over vectors, thus, we omit its explicit and full definition.

We end the *CAO_matrix* theory, and the formalisation of the CAO type system in EasyCrypt, with the proof of some relevant lemmas about matrices.

$$
\begin{array}{rl}
\forall m1, m2 : CAO\_matrix, & m1 = m2 \iff \\
& \mathbf{CAO\_matrix\_size}\ m1 = \mathbf{CAO\_matrix\_size}\ m2 \wedge \\
& \forall i, j : CAO\_int, 0 \leq i < \mathbf{fst}\ (\mathbf{CAO\_matrix\_size}\ m1) \wedge \\
& 0 \leq j < \mathbf{snd}\ (\mathbf{CAO\_matrix\_size}\ m1) \Rightarrow \\
& m1.[(i,j)] = m2.[(i,j)] \\
\forall m : CAO\_matrix, i, j : CAO\_int, x : t, & 0 \leq i < \mathbf{fst}\ (\mathbf{CAO\_matrix\_size}\ m) \Rightarrow \\
& 0 \leq j < \mathbf{snd}\ (\mathbf{CAO\_matrix\_size}\ m) \Rightarrow \\
& (m.[(i,j) \leftarrow x]).[(i,j)] = x \\
\forall m : CAO\_matrix, i1, j1, i2, j2, : CAO\_int, x : t, & 0 \leq i1 < \mathbf{fst}\ (\mathbf{CAO\_matrix\_size}\ m) \Rightarrow \\
& 0 \leq j1 < \mathbf{snd}\ (\mathbf{CAO\_matrix\_size}\ m) \Rightarrow \\
& 0 \leq i2 < \mathbf{fst}\ (\mathbf{CAO\_matrix\_size}\ m) \Rightarrow \\
& 0 \leq j2 < \mathbf{snd}\ (\mathbf{CAO\_matrix\_size}\ m) \Rightarrow \\
& i1 \neq i2 \vee j1 \neq j2 \Rightarrow \\
& (m.[(i1,j1) \leftarrow x]).[(i2,j2)] = m.[(i2,j2)] \\
\forall m : CAO\_matrix, & \mathbf{fst}\ (\mathbf{CAO\_matrix\_size}\ m) = 1 \Rightarrow \\
& \mathbf{length}(\mathbf{to\_array}\ m) = \mathbf{snd}\ (\mathbf{CAO\_matrix\_size}\ m) \\
\forall m : CAO\_matrix, i : CAO\_int, & \mathbf{fst}\ (\mathbf{CAO\_matrix\_size}\ m) = 1 \Rightarrow \\
& 0 \leq i < \mathbf{snd}\ (\mathbf{CAO\_matrix\_size}\ m) \Rightarrow \\
& (\mathbf{to\_array}\ m).[i] = m.[(i,0)]
\end{array}
$$

6.4 CAO TO EASYCRYPT MAPPING ALGORITHM

Our CAO to EasyCrypt mapping algorithm was designed with two main objectives:

- To correctly map every CAO structure and command into some piece of EasyCrypt code that has the same meaning.
- To correctly map every annotation and logic specification presented in the CAO code into EasyCrypt. This implies the generation of the necessary lemmas to prove the program safety and some functional property that may have been specified.

6.4.1 *Preprocessing*

Before starting to produce an EasyCrypt script from a CAO implementation, we perform some preprocessing over data, so that we can improve the efficiency of the translation process. The abstract syntax trees of the CAO language and of the *pWhile* language available in EasyCrypt are very similar, thus the abstract syntax tree that results from the syntactic and semantic analysis could be used to produce the EasyCrypt script. However, we followed a different approach and created another logic structure, that results from the abstract syntax tree but is independent from it, and that allows a better organisation when producing the EasyCrypt script.

Informally, this preprocessing phase creates a series of lists that contain information about the elements of the CAO program. We chose lists instead of maps because we need to order then in order to avoid dependency errors. The information about a CAO program that is captured during this step is the following:

- Global integer constants - that need to be defined first because, for example, potentially many type declarations will depend on then.
- Other global constants - that need to be defined before the CAO program itself.
- Global variables - this list is important due to syntactic restrictions of EasyCrypt. Global variables may appear everywhere in the CAO program but they must all be defined inside of the EasyCrypt *module* but before any procedure and with a specific syntax. Additionally, they may not be initialised, thus, this structure helps the creation of an *init* function, that initialises all the global variables to its value.
- Local variables - again, it is important to collect the local variables of every function due to EasyCrypt syntactic restrictions because variable declaration must be made before any instruction in some EasyCrypt procedure. Note that control variables used in the *seq* loops are also considered local variables.
- Bit string literals - EasyCrypt does not contemplate the definition of bit string literals and, therefore, in order to be used in the EasyCrypt script, every bit string that appears in the CAO program needs to be initialised in the *init* function.
- Functions - since they are an essential element of every CAO program, we also specify a list that contains information about all the functions of the program. This information contemplates the function arguments, its statements and its possible contract.
- Logic nodes - we also collect information about the logic specifications that appear in the CAO program, so that they can be translated before the definition of the CAO program in some EasyCrypt *module*.

In this section, we will focus on the description of how the CAO code is translated to EasyCrypt without considering any CAO-SL structure. A description of how the annotations and logic specifications are mapped can be found in Section 6.5. We will abstract the CAO syntax and consider its abstraction and representation in the logic structure described above.

NOTATION We will use the symbol $\mapsto$ to denote the mapping of a logic representation of a CAO element to the corresponding EasyCrypt syntax. Elements of the CAO language will be represented by its componentes in a tuple form $(x_1 \ * \ ... \ * \ x_n)$. For example, a variable in CAO is represented by its identifier id, its type τ and its value v using the form $(id \ * \ \tau \ * \ v)$. We will use the syntax $None$ and $Some\ x$ to denote some optional parameter x.

6.4.2 *Global integer constants*

The first step of the translation process is to identify all the global integer constants present in the CAO program - both *int* and *register int* constants. This first step is extremely important because many of the types used in the remaining CAO program will depend on these constants, like, for example, some vector or some ring type. Additionally, integer constants may also depend on one another, which requires that the constants appear in the EasyCrypt script with the same order of the CAO program.

At this point, the typechecker, with the preprocessing phase, has already produced a structure containing all the constants of the CAO program. Therefore, being *id* the name of the constant, *t* its type and *e* its possible value or condition, a constant is syntactically mapped as represented in Figure 43.

(id * t * None) ↦ **const** id : t. *(∗No initial value ∗)*
(id * t * Some e) ↦ **const** id : t = e. *(∗Initial value e ∗)*
(id * t * Some e) ↦ **const** id : t. **axiom** id_cond : e. *(∗Condition e ∗)*

Figure 43: Constant mapping

The remaining global constants are translated after the cloning of the types of the program, using the same algorithm of Figure 43.

6.4.3 *Type cloning*

The type cloning phase contemplates the cloning of every type used in the CAO program according to the theories of the CAO types described in Section 6.3. Naturally, the integer and boolean types are not cloned, since they do not need any step of initialisation. The output produced by this step is simply a list of EasyCrypt clonings, according to Figure 44.

Integer ↦ *CAO_int*
RInteger ↦ *CAO_reg_int*
Bool ↦ *CAO_int*
Bits (Signed, e) ↦ **clone** *CAO_bits* **as** Signed_CAO_bits_ie **with op** size = ie.
Bits (Unsigned, e) ↦ **clone** *CAO_bits* **as** Unsigned_CAO_bits_ie **with op** size = ie.
Mod (None, None, p) ↦ **clone** *CAO_mod* **as** CAO_mod_p **with** n = p.
Mod (Some t, Some s, p) ↦
clone *CAO_exfield* **as** CAO_exfield_p **with type** coef_type = t, **op** coef_type_zero = t.CAO_mod_zero.
Vector (e, t) ↦ **clone** *CAO_vector* **as** CAO_vector_t_e **with op** size = e, **type** x = t.
Matrix (e1, e2, t) ↦
clone *CAO_matrix* **as** CAO_matrix_t_e1_e2 **with op** n_rows = e1, **op** n_columns = e2, **type** x = t.

Figure 44: Type mapping

In Figure 44, the left side shows the representation of CAO types as they are used in the typechecking rules of Section 3.2. We provide a brief description of that type representation.

- *Integer* - arbitrary precision integers.

- $RInteger$ - integers in the range $[0, 2^w[$.
- $Bits(Signed, e)$ - signed bit strings with size e.
- $Bits(Unsigned, e)$ - unsigned bit strings with size e.
- $Mod(None, None, p)$ - ring or field $\mathbb{Z}_e$ type.
- $Mod(Some\ t, Some\ s, p)$ - extension field $\mathbb{Z}_t[s]$ type, with s being the variable used in the polynomial and t the type of the coefficients.
- $Vector(e, t)$ - vector of type t and of length e.
- $Matrix(e1, e2, e3)$ - matrix of type t with $e1$ rows and $e2$ columns.

Structures are defined as a tuples and operations are created using the names of the elements of the structure. We illustrate the definition of structures with an example. Suppose the following structure, that specifies the key type of the RC4 encryption scheme.

```
typedef BYTE := unsigned bits[8];

typedef RC4_KEY := struct [
  def x : BYTE;
  def y : BYTE;
  def data : vector[256] of BYTE;
];
```

The $RC4\_KEY$ structure will be abstractly defined as the following tuple

```
type RC4_KEY = Unsigned_CAO_bits_8.CAO_bits * Unsigned_CAO_bits_8.CAO_bits *
    CAO_vector_Unsigned_CAO_bits_8_256.CAO_vector.
```

In order to reason about the elements of the structure, the translation process will also generate three operations, one per each element of the structure.

```
op RC4_KEY_x (t: RC4_KEY) : Unsigned_CAO_bits_8.CAO_bits = t.`1.
op RC4_KEY_y (t: RC4_KEY) : Unsigned_CAO_bits_8.CAO_bits = t.`2.
op RC4_KEY_data (t: RC4_KEY) : CAO_vector_Unsigned_CAO_bits_8_256.CAO_vector = t.`3.
```

6.4.4 *A CAO program as an EasyCrypt module*

EasyCrypt ambient environment do not support the definition of imperative procedures. Therefore, a *module* is created, to support the specification of the remaining elements of the CAO program. This *module* has the same name as the CAO file.

GLOBAL VARIABLES Global variables could be translated to the EasyCrypt script at the same time as global constants, using the **op** EasyCrypt command. For instance, an integer global variable x with some value e could be defined as follows.

```
op x : int = e.
```

However, this option would limit the possible changes to the variable value. Thus, since we do not want variables to behave as constants, we collect all global variables and define then at the beginning of the module, before the procedures. The algorithm to translate global variables can be found in Figure 45.

(id * t * None) $\mapsto$ **var** id : t
(id * t * Some e) $\mapsto$ **var** id : t

Figure 45: Global variables mapping

Note that, even if some global variable as some initial value, it is not assigned to it when it is declared. This is an EasyCrypt syntactic restriction which is solved by defining a new function - *init* - where global variables are initialised. The *init* function takes no input, produces no output and its body is filled with assignment statements. Consequently, for every global variable to which some initial value is assigned, there will be an EasyCrypt assignment command in the *init* function, as described in Figure 46.

$\forall$ (id * t * Some e), (id * t * Some e) $\mapsto$ id = e;

Figure 46: Global variables initialisation mapping

STRUCTURES AND STRUCTURE FIELDS We have seen in Section 6.4.3 how the structure types are defined in EasyCrypt. However if one needs to assigne some value to a structure field, EasyCrypt does not support the use of functions as left values. Therefore, the structure and its fields are declared as global variables and, in the *init* function, there is an assignment statement with different semantics of the others: instead of attributing a value to some variable, it explicitly define the elements of some tuple. Considering the structure of the example in Section 6.4.3 and a variable *key* of the type $RC4\_KEY$, the structure is initialised as follows.

```
var x : Unsigned_CAO_bits_8.CAO_bits

var y : Unsigned_CAO_bits_8.CAO_bits

var data : CAO_vector_Unsigned_CAO_bits_8_256.CAO_vector

var key : RC4_KEY

proc init() : unit = {
    (x,y,data) = key;
}
```

Formally, the algorithm that translates variables with structure type declaration is presented in Figure 47. The notation $(id * sfs)$ describes a structure (with identifier id and fields sfs), $(id * t)$ describes a structure field (with identifier id and type t) and to_tuple is an auxiliary function that generates tuples from the fields of the structure with the representation $(id_1, ..., id_2)$.

$$\forall \text{ (id * t)} \in Struct, \text{(id * t)} \mapsto \textbf{var id : t}$$
$$\wedge$$
$$\text{(id * sft)} \mapsto \text{to\_tuple sfs = id;.}$$

Figure 47: Global variables with structure type initialisation mapping

BIT STRING LITERALS EasyCrypt does not support the explicit description of bit strings literals like CAO. Thus, in order to be able to use them like one could use in CAO expressions, there is the need to initialise them following their representation in the *CAO_bits* theory - as EasyCrypt *bitstring*s, which, by their turn, are formalised as boolean arrays. Bit strings are also declared as global variables and initialised in the *init* function.

Since there is no identifier attached to literals, bit string literals are declared as variables with name bs_i, where i is a number between 0 and the number of bit string literals in the CAO program. The bit strings are then initialised in the *init* function as boolean arrays. The formal algorithm that describes this translation can be found in Figure 48.

$$\text{(bs * t)} \mapsto \textbf{var bs\_i : t}$$
$$\wedge$$
$$\forall j \in [0..length\ bs], bit \in fst\ \text{(bs * t)}, \text{(bs * t)} \mapsto \text{bs\_i = bs\_i[j <- bit];}$$

Figure 48: Bit string literals mapping

PROCEDURES/FUNCTIONS CAO functions are mapped into EasyCrypt procedures, only defined through *modules*. From the preprocessing phase, we are in possession of a sorted list containing all the functions of the CAO program, including their arguments, statements and possible contracts.

The elements of the list are of the form $(id * t * ctr * args * stmts)$, with id the name of the function, t its return type, ctr its contract, $args$ its list of arguments and $stmts$ its list of statements. Arguments are structured as $(id_a * t_a)$, with id the name of the argument and t its type.

The translation of a CAO function to an EasyCrypt procedure starts by producing the header of the function according to the EasyCrypt syntax. The algorithm works accordingly to function Figure 49.

(id * t * ctr * args * stmts) $\mapsto$ **proc** id ($\forall$ (id_a * t_a) $\in$ *args*, (id_a * t_a) $\mapsto$ id_a : t) : t

Figure 49: Function header mapping

Recalling Section 6.2.2, we added the support to parametrise CAO functions by constant parameters, attached with some condition. When mapped to EasyCrypt, the constant parameters of a function are considered normal arguments of the function and the conditions of the parameters are added to the contract of the function, as preconditions. This can be done without loss of generality because the program has been previously typechecked.

After the declaration of the header of the function, the next step to take is to declare all the local variables. Local variables need to be declared before any instruction due to EasyCrypt syntactic restrictions. The algorithm to translate local variables can be found in Figure 50. Naturally, the local variables structure has a parameter f that concerns the function where variables are declared.

(id * t * f * None) $\mapsto$ **var** id : t;
(id * t * f * Some e) $\mapsto$ **var** id : t = e;

Figure 50: Local variables mapping

Finally, the translation of CAO functions gets completed with the generation of the remaining commands (other than declarations). There are some similarities between the CAO and the *pWhile* syntaxes that could transform this translation into a direct syntactic translation. The only two exceptions are the sampling command and the *seq* loop. In what respects the first one, to perform some sampling operation in *pWhile* language, there needs to be an explicit description of the probability distribution, whereas in CAO, the distributions are uniform over the support of the type of the left value (recall Section 6.2.2). In what respects the last one, there is no instruction in *pWhile* similar to the *seq* CAO statement, which requires a conjunction of other commands that results in a block of code with the same semantics of *seq* loops.

The recursive mapping algorithm that produces function statements can be consulted in Figure 51. Commands are represented by the abstraction that is used in the typechecking process, with the following meaning:

- $Sample(lval)$ - sampling from an uniform distribution of the type of $lval$, being the result stored in $lval$. The probability distributions are defined in the type EasyCrypt theory.
- $FCallS(f, es)$ - calling function f with the list of arguments es.
- $Ret(e)$ - returns the value of expression e.

- $Ite(e, stmts, None)$ - conditional statement without the *else* branch, with condition e and $stmts$ being the statements that compose the *if* branch.
- $Ite(e, stmts1, Some\ stmts2)$ - conditional statement with the *else* branch, with condition e, $stmts1$ being the statements that compose the *if* branch and $stmts2$ being the statements that compose the *else* branch.
- $Seq(ctr, SeqIter(s, e1, e2, None), stmts)$ - *seq* loop, with possible contract ctr, control variable s, starting condition $e1$, ending condition $e2$ and composed by statements $stmts$.
- $Seq(ctr, SeqIter(s, e1, e2, Some\ e3), stmts)$ - *seq* loop, with possible contract ctr, control variable s, starting condition $e1$, ending condition $e2$, with an incrementation rate of s defined by $e3$ and composed by statements $stmts$.
- $While(ctr, e, stmts)$ - *while* loop, with possible contract ctr, loop condition e and composed by statements $stmts$.

Assign(lval, e) $\mapsto$ lval = e;
Sample (lval) $\mapsto$ lval $\xleftarrow{\$}$(get_type lval).distr
FCallS (f, es) $\mapsto$ f (es);
Ret (e) $\mapsto$ **return** e;
Ite (e, stmts, None) $\mapsto$ **if** (e){ stmts }
Ite (e, stmts1, Some stmts2) $\mapsto$ **if** (e){ stmts1 } **else** { stmts2 }
Seq (ctr, SeqIter(s, e1, e2, None), stmts) $\mapsto$ s = e1; **while** (s <> e2){ stmts; s = s + 1; }
Seq (ctr, SeqIter(s, e1, e2, Some e3), stmts) $\mapsto$ s = e1; **while** (s <> e2){ stmts; s = s + e3; }
While (ctr, e, stmts) $\mapsto$ **while** (e){ stmts }

Figure 51: Statement mapping

EXPRESSIONS Expressions in CAO are translated to EasyCrypt in an almost direct translation. The only detail worth mentioning is the fact that all the operators follow a prefix style, in order to overcome some difficulties of the EasyCrypt typechecker to deal with operators that may be override to different types. The prefix style implies the explicit identification of the types involved in the operation, which eliminates typechecking errors. For example, an addition operation between two inhabitants of the type $\mathbb{Z}_7$ - x and y - would be translated to EasyCrypt as follows.

x + z $\mapsto$ CAO_mod_7.(+)x y

In what concerns operations over CAO integers and CAO booleans, the prefix theories that define the operations are, respectively, the EasyCrypt *Int* and *Bool* theories. Expressions in logic specifications are mapped following an infix style.

6.5 CAO-SL TO EASYCRYPT MAPPING ALGORITHM

Annotations to the CAO program are very important in what concerns deductive verification. They allow the specification of pre- and postconditions, loop invariants and variants and also some logic elements that can be used to help when discharging proofs.

6.5.1 *Logic specifications*

Our mapping algorithm translates every logic specification to EasyCrypt before the generation of the program. This organisation is important because some logic definition may be used in, for example, a program assertion.

CONSTRUCTOR at The constructor at is used to refer to the value of some expression in some particular state in the execution of the code, defined by a label. In EasyCrypt, we are able to reason about memories of programs but not about its states, i.e., we are able to reason about the value of expressions that belong to a program but not to reason about the value of expressions in states of the program (besides the pre-state and the post-state). Consequently, every occurence of the at constructor will be translated to the EasyCrypt by simply translating the expression to which the at constructor is applied.

LOGIC FUNCTIONS Logic functions in CAO-SL can appear with or without definition and, according to this characteristic, the translation will be different. The algorithm that describes the process can be found in Figure 52, where to_tuple' is a function that generates tuples with the representation $(t\_a_1 * \ldots * t\_a_n)$ and the function to_tuple'' is a function that generates tuples with the representation $(id\_a_1 : t\_a_1, \ldots, id\_a_n : t\_a_n)$.

(id * t * args * None) $\mapsto$ **op** id : to_tuple' args $\rightarrow$ t.
(id * t * args * Some e) $\mapsto$ **op** id (to_tuple'' *args*): t = e.

Figure 52: Logic functions mapping

For example the logic function `logic int sum{L} (a,b : int) = a + b` would be mapped into **op** sum (a, b : *CAO_int*): *CAO_int* = a + b, whereas the logic function `logic int sum' {L} (a,b : int)` would be mapped into **op** sum' : *CAO_int* $\rightarrow$ *CAO_int* $\rightarrow$ *CAO_int*.

PREDICATES Predicates in CAO-SL are translated to EasyCrypt in a very similar way to logic functions. In fact, if only the header of the predicate is provided, we consider a predicate as a function with a boolean return value. The only differences -in the case of a body to the predicate is provided - are the use of the **pred** keyword (instead of the **op** keyword) and the absence of a return type (since it is always a boolean value). The translation algorithm for predicates is represented in Figure 53.

(id * args * None) $\mapsto$ **op** id : to_tuple' args $\rightarrow$ *CAO_bool*.
(id * args * Some e) $\mapsto$ **pred** id (to_tuple'' *args*)= e.

Figure 53: Predicates mapping

For instance, the CAO-SL predicate `predicate equa{L1,L2}(u,v : unsigned bits[10]) = forall l:int; 0 <= l < 10 ==> at(u[l],L1)== at(v[l],L2)` would be mapped into **pred** equal (u v:Unsigned_CAO_bits_10.*CAO_bits*)= $\forall$(l:*CAO_int*), 0 $\leq$l $<$ 10 $\Rightarrow$ u[l] = v[l], whereas the predicate `predicate equal'{L1,L2}(u,v : unsigned bits[10])` would be translated into **op** equal' : Unsigned_CAO_bits_10.*CAO_bits* $\rightarrow$ Unsigned_CAO_bits_10.*CAO_bits* $\rightarrow$ *CAO_bool*.

INDUCTIVE PREDICATES In CAO-SL, in order to define an inductive predicate, one first defines the predicate (and its arguments) and then provide one (or more) base cases and an inductive case for it. Informally, our translation algorithm defines the predicate as a logic function and then axiomatises its behaviour according to the *cases* of the predicate.

The translation algorithm is described in Figure 54. The inductive cases are represented by the list *ind* and are represented by the structure $(id * e)$.

(id * args * ind) $\mapsto$ **op** id : to_tuple' args $\rightarrow$ *CAO_bool*. $\forall$ (id_i * e) $\in$ *ind*, (id_i * e) $\mapsto$ **axiom** id_i : e.

Figure 54: Inductive predicates mapping

LEMMAS Lemmas are very useful when performing proofs. For example, they provide a mean to reduce the complexity of some proof: by proving an auxiliary proposition using a lemma, some other proof may become trivial. In CAO-SL, lemmas are defined without any proof script, which could represent a difficulty when translating the lemma to EasyCrypt. Nevertheless, since they are first-order logic propositions, the mapping of CAO-SL lemmas to EasyCrypt contemplates the specification of a simple proof script that consists only on an SMT call. The algorithm that deals with the translation of CAO-SL lemmas is presented in Figure 55.

(id * p) $\mapsto$ **lemma** id : p **by smt**.

Figure 55: Lemmas mapping

As an example, the lemma `transitivity` of Section 6.5.1, would result in the following EasyCrypt code.

```
lemma transitivity : ∀(u v z: Unsigned_CAO_bits_10.CAO_bits), equal u v ∧equal v z ⇒ equal u z
    by smt.
```

AXIOMATIC DEFINITIONS An *axiomatic* is a logic definition that contemplates logic types and predicates and operations over those types. Therefore, the translation of an axiomatic is defined recurring to the previously defined translations. The only detail worth mentioning is the possibility to define new logic types inside the axiomatic. These types are easily translated from CAO-SL to EasyCrypt through the definition of a new type with the keyword **type**.

We provide an example to illustrate the mapping of CAO-SL axiomatic definitions. Consider the *lists_axiomatic* in Section 6.5.1. The translation process should be able to define a new logic type *list*, a logic element of type *list*, two logic operations over the *list* type and two axioms. The resulting EasyCrypt script follows.

```
type list.
op nil : list.
op append : list * list -> list.
op cons : CAO_int * list -> list.
axiom append_nil: ∀(l:list), append(l,nil) = l.
axiom append_cons: ∀(l1 l2:list) (n:CAO_int), append(cons(n,l1),l2) = cons(n,append(l1,l2)).
```

6.5.2 *Ghost code*

EasyCrypt does not support the definition of ghost code. Therefore, any ghost code annotation will not be mapped into the EasyCrypt script.

6.5.3 *Function contracts*

The EasyCrypt *pWhile* programming language does not support any logic specification other than assertions. In order to reason about a function and its contract, EasyCrypt allows the specification of Hoare triples. The function contracts, introduced using CAO-SL, will be added to the EasyCrypt script in the form of propositions in the Hoare triple.

For every annotated function present in the CAO program, one lemma to prove the validity of the respective Hoare triple is generated. The algorithm to create these lemmas is showed in Figure 56, where function *get_pre* takes as input a function contract and outputs its precondition, function *get_post* takes as input a function contract and outputs its postcondition.

EasyCrypt provides a keyword - **res** - to refer to the value that is outputted by the function. Yet, in order to reason about values in the precondition in the postcondition, EasyCrypt does not contemplate any mechanism or keyword for it. To overcome this difficulty, we define the function lemma as a quantification on as many logic variables as input variables and then add to the precondition clauses that attest the equality between the input variables and the logic variables. Consequently, by referring

to the these logic variables, we are actually referring to variables that have the old value of the input variables.

An example of the generation of Hoare triple lemmas about functions follows. Suppose a CAO function that takes as input two integers values and that outputs the addition of both values. It would be mapped into the following EasyCrypt function.

```
proc sum(x : CAO_int, y : CAO_int ) : CAO_int = {

    return (x+y);
}
```

Suppose now that the function is annotated with the precondition $true$ and with postcondition $res = old(x) + old(y)$. The corresponding EasyCrypt lemma would be defined as presented next.

```
lemma sum (x' y' : CAO_int): hoare [sum : x = x' ∧y = y' ∧true ==> res = x' + y'].
```

$$
\begin{gathered}
\text{(id * t * ctr * args * stmts)} \mapsto \textbf{lemma id\_hoare (gen\_logic\_vars args) : hoare [id : get\_pre ctr /} \\
\text{add\_eqs args ==> get\_post ctr].} \\
\wedge \\
\forall\ \text{(id\_a * t\_a)} \in \text{args, (id\_a * t\_a)} \mapsto \text{id\_a' : t\_a} \\
\wedge \\
\forall\ \text{(id\_a * t\_a)} \in \text{args, (id\_a * t\_a)} \mapsto \text{id\_a' = id\_a}
\end{gathered}
$$

Figure 56: Function contracts mapping

In contrast to the previous version of CAOVerif, the backend tool does not provide the same automation degree. However, we generate a proof script for functions, in order to prove the validity of the Hoare triple related to it. We refer the reader to Section 6.5.5 for a better understanding of how this proof script is generated.

6.5.4 *Statement annotations*

CAO-SL allows two types of statement annotations - assertions and loop annotations - but only the first ones can be specified directly in a program written in the *pWhile* language. Assertions in EasyCrypt are written exactly as they are written in CAO-SL: being p a predicate, an assertion is written as **assert** p;.

Loop annotations are translated directly into the proof script using the **while** tactic. Note that, as presented in Section 5.2.3, in Hoare logic, the **while** EasyCrypt tactic only takes as input the invariant, since no termination proof is generated by the EasyCrypt proof engine. Let inv be the loop invariant specified by some user, it will appear in the generated proof script as **while** (inv).

6.5.5 *A proof script*

EasyCrypt is an interactive proof assistant, meaning that one needs to *guide* the tool in order to complete some proof. This is the exact opposite of what happens with the backend tool of the previous version of CAOVerif. Using the Frama-C/Jessie toolchain, the program was parsed through an automatic verification conditions generator, which generated all the proof obligations related to some program. In EasyCrypt, proof obligations are generated iteratively, according to the tactic applied. Consequently, it is hard to be certain about how a proof script will conduct. For example, in the presence of an assignment operation, the most intuitive option was to apply the **wp** tactic so that the postcondition was updated in respect to the assignment. However, one could want to weaken the precondition with the **conseq** tactic first to deal better with the remaining proof obligations. Nevertheless, it is possible to generate a proof script that would fit the proof in question. In this section we describe the generation of proof scripts for the lemmas that aim to reason about functions and its contracts. These scripts are very simple and elementary yet, besides not contemplating any advanced proof strategy or tactic, they permit the proof of an interesting set of lemmas.

We defined an association between CAO statements and EasyCrypt tactics to be used in the presence of those statements. This relation is presented in Figure 57. Note that there is no tactic defined for the return statement since it is automatically consumed by the proof environment. We use the same symbol ($\mapsto$) to define this relation and we define the recursive application of this relation with the function *get_tactics*, that, given a list of CAO statements, finds the suitable EasyCrypt according to the relation of Figure 57. Auxiliary functions *get_ctr* and *get_inv* return the contract of a function and the invariant of a loop, respectively.

Assign(lval, e) $\mapsto$ **wp**.
Sample (lval) $\mapsto$ **rnd**.
FCallS (f, es) $\mapsto$ **call** (_ : get_pre (get_ctr f) ==> get_post (get_ctr f)).
Ite (e, stmts, None) $\mapsto$ **wp**.
Ite (e, stmts1, Some stmts2) $\mapsto$ **wp**.
Seq (ctr, SeqIter(s, e1, e2, None), stmts) $\mapsto$ **while** (get_inv ctr). get_tactics stmts.
Seq (ctr, SeqIter(s, e1, e2, Some e3), stmts) $\mapsto$ **while** (get_inv ctr). get_tactics stmts.
While (ctr, e, stmts) $\mapsto$ **while** (get_inv ctr). get_tactics stmts.

Figure 57: Relation between CAO statements and EasyCrypt tactics

The algorithm that generates proof scripts for function reasoning works by obtaining the function statements - which can be done efficiently using the data structure outputted by the preprocessing phase (Section 6.4.1) - and then by applying the algorithm of Figure 57 to them. The algorithm finishes by applying the **skip** tactic and then by calling external SMT solvers using the **smt** tactic.

We state that a proof script produced using the presented algorithm suits every function in which conditional blocks are composed by deterministic straightline code without procedure calls. Informally, this means that inside an `if` statment there can only be assignments or other deterministic `if`

statements. In order to be able to deal with every class of programs, there was the need to use the **case**, **rcondt** and **rcondf** tactics in an interleaved way. This fact would require, for example, that we keep track of the line of the `if` statements in the EasyCrypt proof environment, which represents information that we are not able to obtain and that could compromised the desired levels of efficiency.

6.6 SAFETY PROPERTIES

Safety is related with preventing runtime errors which are due to not accounted situations in the evaluation semantics of the programming language. The inference system for these safety-sensitive Hoare triples (already defined in Figure 8) does not depend only on the structure of the command, but also on expressions that may occur in it. The side-conditions of each rule include special conditions, called *safety conditions*, whose validity implies that the program does not evaluate to an error state.

In this section, we present how we delt with safety proofs in our renewed CAOVerif tool. Intuitively, the process works by generating a new EasyCrypt *module* to which the CAO program will be translated exactly like presented in Section 6.4.4. However, every statement will be annotated with an assertion that contains safety conditions. Using this safety-sensitive EasyCrypt procedure and the ability to emulate Hoare triples, we were able to model safety-sensitive Hoare triples and, by proving their validity, prove the safety of the function they concern.

6.6.1 *The safe predicate in EasyCrypt*

In order to be able to introduce safety properties in assertions, we defined a *safe* predicate that reasons about inputs of an arbitrary type. This way, we can apply the *safe* predicate to all CAO types previously formalised. The behaviour of the *safe* predicate is axiomatised for each CAO type in its respective theory. Table 5 sums up the safety proof obligations that are generated for each type.

CAO_INT SAFETY The safety of integer operations was slightly introduced in Figure 9. Since we consider arbitrary precision integers, our focus lies only on the division and modulo operation. Yet, since in CAO there is no type *real*, one can not perform exponentiation with negative values, since it would result in a fractional number. We added a new safety condition to encounter this CAO restriction: the exponents in the exponentiation operation needs to be greater or equal to zero.

$$\mathbf{safe}(e_1 ** e_2) = \mathbf{safe}(e_1) \wedge \mathbf{safe}(e_2) \wedge e_2 \geq 0$$

CAO_REG_INT SAFETY The safety conditions of operations involving elements of the register int CAO type are equal to the safety conditions defined above for the *CAO_int* type. However, when dealing with bounded integers, we need to consider possible overflows. This safety property is captured by the following restriction

Type	Operation	Proof obligation
int	$e_1 / \ e_2$	$e_2 \neq 0$
	$e_1 \% \ e_2$	$e_2 > 0$
	$e_1 ** e_2$	$e_2 \geq 0$
register int	$e_1 / \ e_2$	$e_2 \neq 0$
	$e_1 \% \ e_2$	$e_2 > 0$
	$e_1 ** e_2$	$e_2 \geq 0$
	$e_1 \circ e_2$	$e_1 \circ e_2 < 2^w$, where $\circ \in \{+, -, *, /, \%, **\}$
mod[n]	$e_1 / \ e_2$	$gcd(toint\ e_2, n) = 1 \wedge e_2 \neq CAO\_mod\_zero$
	$e_1 ** e_2$	$e_2 \geq 0$
mod[$\tau \ < X > / p(X)$]	$e_1 / \ e_2$	$e_2 \neq CAO\_exfield\_zero$
	$e_1 ** e_2$	$e_2 \geq 0$
signed/unsigned bits[n]	$b[e]$	$0 \leq e < n$
	$b[e_1 \ldots e_2]$	$0 \leq e_1 < e_2 < n$
	$b << e,\ b >> e$	$0 \leq e < n$
	$b <\mid e,\ b \mid > e$	$0 \leq e < n$
vector[n] of τ	$v[e]$	$0 \leq e < n$
	$v[e_1 \ldots e_2]$	$0 \leq e_1 < e_2 < n$
	$v << e,\ v >> e$	$0 \leq e < n$
	$v <\mid e,\ v \mid > e$	$0 \leq e < n$
matrix[n_1, n_2] of τ	$m[e_1, e_2]$	$0 \leq e_1 < n_1 \wedge 0 \leq e_2 < n_2$
	$m[e_1 \ldots e_2, e_3]$	$0 \leq e_1 < e_2 < n_1 \wedge 0 \leq e_3 < n_2$
	$m[e_1, e_2 \ldots e_3]$	$0 \leq e_1 < n_1 \wedge 0 \leq e_2 < e_3 < n_2$
	$m[e_1 \ldots e_2, e_3 \ldots e_4]$	$0 \leq e_1 < e_2 < n_1 \wedge 0 \leq e_3 < e_4 < n_2$
	$m ** e$	$e \geq 0$

Table 5: Safety proof obligations

$$\begin{aligned}\mathbf{safe}(e_1 \circ e_2) = \mathbf{safe}(e_1) \wedge \mathbf{safe}(e_2) \wedge \mathbf{safe}(e_1 \circ e_2) \wedge e_1 \circ e_2 < 2^w,\\ where \circ \in \{+, -, *, /, \%, **\}\end{aligned}$$

CAO_BOOL SAFETY Boolean safety was also introduced in Figure 9 in the form of comparison between integers. Nevertheless, we need to account the safety of the conjunction, disjunction, exclusive disjunction and negation boolean operations. The safety properties for these operations are presented in Figure 58.

$$\begin{aligned}\mathbf{safe}(b_1 \vee b_2) &= \mathbf{safe}(b_1) \wedge (\neg b_1 \Rightarrow \mathbf{safe}(b_2)\\ \mathbf{safe}(b_1 \wedge b_2) &= \mathbf{safe}(b_1) \wedge (b_1 \Rightarrow \mathbf{safe}(b_2))\\ \mathbf{safe}(b_1 \oplus b_2) &= \mathbf{safe}(b_1) \wedge \mathbf{safe}(b_2)\\ \mathbf{safe}(\neg b) &= \mathbf{safe}(b)\end{aligned}$$

Figure 58: Safety of boolean operations

CAO_BITS SAFETY In our model, the *CAO_bits* type inherits the properties of the EasyCrypt *bitstring* type, which is defined as a boolean array. Therefore, when considering accesses to bit strings, the safety properties will be the same introduced in Section 2.6.2.1. The remaining safety conditions are axiomatised as shown in Figure 59.

$$\begin{aligned}\mathbf{safe}(bs) &= true\\ \mathbf{safe}(bs[e]) &= \mathbf{safe}(bs) \wedge \mathbf{safe}(e) \wedge 0 \leq e < \mathbf{length}\ bs\\ \mathbf{safe}(bs[e_1 \ldots e_2]) &= \mathbf{safe}(bs) \wedge \mathbf{safe}(e_1) \wedge \mathbf{safe}(e_2)\ \wedge 0 \leq e_1 \leq e_2 < \mathbf{length}\ bs\\ \mathbf{safe}(bs \circ e) &= \mathbf{safe}(bs) \wedge \mathbf{safe}(e) \wedge 0 \leq e < \mathbf{length}\ bs, where \circ \in \{<<, >>, <|, |>\}\\ \mathbf{safe}(bs_1 \dagger bs_2) &= \mathbf{safe}(bs_1) \wedge \mathbf{safe}(bs_2), where \dagger\ \in \{\wedge, \vee, \oplus, @\}\\ \mathbf{safe}(\neg bs) &= \mathbf{safe}(bs)\end{aligned}$$

Figure 59: Safety of bit string operations

CAO_MOD SAFETY The inhabitants of the CAO $mod[n]$ type are a subset of the inhabitants of the integer type and, thus, the safety conditions for the integer type also apply to the *CAO_mod* type. There is only one simple detail that needs to be taken into account: when performing the division, it is not enough to ensure that the divisor is different than zero, since both operations are only defined under the condition that n and the divisor. This safety property is encompassed by the following axiom.

$$\mathbf{safe}(e_1/e_2) = \mathbf{safe}(e_1) \wedge \mathbf{safe}(e_2) \wedge e_2 \neq 0 \wedge \mathbf{gcd}(\mathbf{toint}\ e_2,\ n) = 1$$

CAO_EXFIELD SAFETY *CAO_exfield* type corresponds to the CAO extension field type $\mathbb{Z}_n[X]$. The operations defined for this type are the typical arithmetic operations: addition, subtraction, unary subtraction, multiplication, exponentiation and division. Consequently, the safety proof obligations will be the same as the other types that support arithmetic operations (*CAO_int* and *CAO_mod*).

CAO_VECTOR SAFETY Safety properties for arrays have been introduced in Section 2.6.2.1 and they mostly concern the prevention of performing accesses out of the bounds of the array. We extend those safety conditions with ones that give respect to other array operations defined in CAO and present in our formalisation of the CAO typesystem, that can be consulted in Figure 60. Note that these safety conditions are similar to the ones that refer to bit strings (Figure 59), except that are applied to the *CAO_vector* type.

$$
\begin{aligned}
\mathbf{safe}(a) &= \mathit{true} \\
\mathbf{safe}(a[e]) &= \mathbf{safe}(a) \wedge \mathbf{safe}(e) \wedge 0 \leq e < \mathbf{length}\ a \\
\mathbf{safe}(a[e_1 \ldots e_2]) &= \mathbf{safe}(a) \wedge \mathbf{safe}(e_1) \wedge \mathbf{safe}(e_2)\ \wedge 0 \leq e_1 \leq e_2 < \mathbf{length}\ a \\
\mathbf{safe}(a \circ e) &= \mathbf{safe}(a) \wedge \mathbf{safe}(e) \wedge 0 \leq e < \mathbf{length}\ a,\ \mathit{where} \circ \in \{<<, >>, <|, |>\}
\end{aligned}
$$

Figure 60: Safety of vector operations

CAO_MATRIX SAFETY The safety proof obligations for the matrix type will be similar to the ones already defined for *CAO_vector* type. Every safety condition to non-arithmetic operations will be an extension of the safety conditions already define for the CAO vector type to two dimensions. The formalisation of the *CAO_matrix* safety conditions can be found in Figure 61.

$$
\begin{aligned}
\mathbf{safe}(m) &= \mathit{true} \\
\mathbf{safe}(m[e_1, e_2]) &= \mathbf{safe}(a) \wedge \mathbf{safe}(e_1) \wedge \mathbf{safe}(e_2) \wedge 0 \leq e_1 < \mathbf{rows}\ m \wedge \\
&\quad 0 \leq e_2 < \mathbf{columns}\ m \\
\mathbf{safe}(m[e_1 \ldots e_2, e_3 \ldots e_4]) &= \mathbf{safe}(m) \wedge \mathbf{safe}(e_1) \wedge \mathbf{safe}(e_2) \wedge \mathbf{safe}(e_3) \wedge \mathbf{safe}(e_4) \wedge \\
&\quad 0 \leq e_1 \leq e_2 < \mathbf{rows}\ m \wedge 0 \leq e_3 \leq e_4 < \mathbf{columns}\ m \\
\mathbf{safe}(m[e_1 \ldots e_2, e_3]) &= \mathbf{safe}(m) \wedge \mathbf{safe}(e_1) \wedge \mathbf{safe}(e_2) \wedge \mathbf{safe}(e_3) \wedge \\
&\quad 0 \leq e_1 \leq e_2 < \mathbf{rows}\ m \wedge 0 \leq e_3 < \mathbf{columns}\ m \\
\mathbf{safe}(m[e_1, e_2 \ldots e_3]) &= \mathbf{safe}(m) \wedge \mathbf{safe}(e_1) \wedge \mathbf{safe}(e_2) \wedge \mathbf{safe}(e_3) \wedge \\
&\quad 0 \leq e_1 < \mathbf{rows}\ m \wedge 0 \leq e_2 \leq e_3 < \mathbf{columns}\ m \\
\mathbf{safe}(m_1 \circ m_2) &= \mathbf{safe}(m_1) \wedge \mathbf{safe}(m_2),\ \mathit{where} \circ \in \{+, -, *\} \\
\mathbf{safe}(-m) &= \mathbf{safe}(m) \\
\mathbf{safe}(m ** e) &= \mathbf{safe}(m) \wedge \mathbf{safe}(e) \wedge e \geq 0
\end{aligned}
$$

Figure 61: Safety of matrix operations

6.6.2 *A safety-sensitive EasyCrypt scheme*

Having a *safe* predicate defined, we could reason about the safety of a CAO program by generating a proof script that was similar to the one presented in Section 6.5.5, with the addition of a **cut** statement for every expression that appeared in the translated CAO program that would reason about the safety of that expression. Nevertheless, we followed a different approach: we generate another EasyCrypt script that is exactly the same except that the statements that compose the procedures inside of the modules are annotated with *assertions* that reason about the safety of the expressions inside the statements.

To produce the new EasyCrypt script with procedures annotated with *assertions*, we define a new translation algorithm for CAO functions, that can be consulted in Figure 62.

For a better illustration of the process, consider the following example of a CAO program that changes the values of a vector according to some condition.

```
def safety_test (x : int) : vector[10] of int {

   def v : vector[10] of int;

   seq i := 0 to 9 {
     if (x % 2 == 0) { v[i] := 1; } else { v[i] := 0; }
   }

   return v;
}
```

Assign(lval, e) $\mapsto$ **assert**(safe e); lval = e;
Sample (lval) $\mapsto$ lval $\xleftarrow{\$}$(get_type lval).distr
FCallS (f, es) $\mapsto$ **assert** ($\forall$ e $\in$ es, safe e); f (es);
Ret (e) $\mapsto$ **assert**(safe e); **return** e;
Ite (e, stmts, None) $\mapsto$ **assert**(safe e); **if** (e){ stmts }
Ite (e, stmts1, Some stmts2) $\mapsto$ **assert**(safe e); **if** (e){ stmts1 } **else** { stmts2 }
Seq (ctr, SeqIter(s, e1, e2, None), stmts) $\mapsto$
assert(safe e1 / safe e2); s = e1; **while** (s <> e2){ stmts; **assert**(safe (s+1)); s = s + 1; }
Seq (ctr, SeqIter(s, e1, e2, Some e3), stmts) $\mapsto$ **assert**(safe e1 / safe e2 / safe e3); s = e1; **while** (s <> e2){ stmts; **assert**(safe (s+e3)); s = s + e3; }
While (ctr, e, stmts) $\mapsto$ **assert**(safe e); **while** (e){ stmts }

Figure 62: *Safety-sensitive* functions mapping

The EasyCrypt script generated would have the following *safety-sensitive* procedure.

```
proc safety_test(x : CAO_int ) : CAO_vector_CAO_int_10.CAO_vector = {
```

```
    var v : CAO_vector_CAO_int_10.CAO_vector;
    var i : CAO_reg_int;

    assert(safe 0);
    i = 0;

    assert (safe (i ≤9));
    while (i ≤9) {

      assert (safe (Int.( %% ) x 2 = 0));
      if (Int.( %% ) x 2 = 0) {

        assert (safe (v[i]) ∧safe 1);
        v[(i)] = 1;
      }

      else {
        assert (safe (v[i]) ∧safe 0);
        v[i] = 0;
      }

      assert (safe (i+1));
      i = i + 1;
    }

    assert (safe v);
    return (v);
}
```

The normal script would be equal to the above described, but without the safety conditions.

6.6.3 *Safety proofs*

The *safety-sensitive* EasyCrypt script is generated as an auxiliary script, so that the original one does not become too big and unreadable. However, the original script *includes* the *safety-sensitive* to perform safety proofs.

In order to prove the safety of some procedure, we prove the equivalence between the original procedure and the procedure annotated with assertions. If the two programs are equivalent, then the CAO procedure meets all the safety restrictions and one is able to continue to perform proofs over that function with confidence that the execution of it will not end in some error state.

The equivalence between the two programs is proven with respect to its annotations. The Hoare triple that represents the equivalence has the form $\{\phi \wedge \theta\} C \sim C_{safe} \{\psi\}$, where ϕ is the precondition for both C and C_{safe}, psi is the postcondition and θ is a new assertion that states that the memories of both programs are the same. Informally, this means that the execution of C and C_{safe} will be the same, despite the fact that for the right side program one also needs to prove the safety proof obligations generated by the *assertion* commands. In EasyCrypt, the $assert(p);$ statement is consumed by the **wp** tactic, that produces a proof goal with p, given the context already built.

A proof scrip is also generated for our safety proofs. For some function f in a module M, the EasyCrypt equivalence lemma is the following.

```
equiv f_safe : M.f ~ M_safety.f : φ ∧ θ ==> ψ.
```

The proof script is equal to the one generated for functional correctness proofs, with the addition of application of the **wp** tactic in the middle of the other tactics, since an assertion condition will always appear. In probabilistic relational Hoare logic proofs in EasyCrypt, the application of a tactic consumes information of both programs. The algorithm to generate the proof script is defined in Figure 63. Note that there is no need to apply the **wp** tactic one second time in the presence of an assignment or a conditional block, since the **wp** tactic is already applied and, due to its recursive behaviour, it is not necessary to apply it a second time.

Assign(lval, e) ↦ **wp**.
Sample (lval) ↦ **rnd**.
FCallS (f, es) ↦ **wp**. **call** (_ : get_pre (get_ctr f)⇛ get_post (get_ctr f)).
Ite (e, stmts, None) ↦ **wp**.
Ite (e, stmts1, Some stmts2) ↦ **wp**.
Seq (ctr, SeqIter(s, e1, e2, None), stmts) ↦ **wp**. **while** (get_inv ctr). get_tactics stmts.
Seq (ctr, SeqIter(s, e1, e2, Some e3), stmts) ↦ **wp**. **while** (get_inv ctr). get_tactics stmts.
While (ctr, e, stmts) ↦ **wp**. **while** (get_inv ctr). get_tactics stmts.

Figure 63: Proof script for safety proofs

6.7 EXAMPLE

In order to illustrate the behaviour of CAOVerif, consider the bellow defined `fact` function, that calculates the factorial of some integer input.

```
def fact(n : int) : int {

  def f : int := 1;
  def i : int := 1;
```

```
  while (i <= n) {
    f := f * i;
    i := i + 1;
  }

  return f;
}
```

Using CAO-SL, one is able to define a `factorial` axiomatic, describing the factorial theory. The axiomatic contemplates an inductive definition of the factorial algorithm, using a logic function `lfact` that explicitly computes the factorial of the input integer based on the inductive properties of the factorial operation.

```
/*@ axiomatic factorial {
  logic int lfact (n:int);
  axiom fact0: lfact(0) == 1;
  axiom factInd: lfact(n) == n * lfact(n-1);
} */
```

Using the `factorial` axiomatic, one is now able to annotate the function with the following contract, that requires the input of the `fact` function to be greater or equal than zero and that its output will be equal to the factorial of the input, according to the `fact` function specified in the axiomatic.

```
/*@ requires n >= 0;
   ensures result == lfact(n); */
```

Finally, the while loop should be annotated with an invariant that correctly represents the incrementation of the control variable `i` and of the value of the variable `f`.

```
/*@ invariant 1 <= i <= n && f = lfact(i-1) */
```

As already stated in the previous section, the execution of CAOVerif over this factorial program would result in two files: *factorial.ec* (with the normal translation of the CAO program) and *factorial_safe.ec* (with the translation that considers safety conditions). The tool would start by mapping the `factorial` axiomatic as follows.

```
op lfact : CAO_int → CAO_int.
axiom lfact0 : lfact 0 = 1.
```

```
axiom lfactInd: ∀(n : CAO_int), lfact n = n ∗ lfact (n−1).
```

The factorial function would then be translated by creating an EasyCrypt module and by defining a new EasyCrypt procedure. The result of the translation of the CAO `fact` is represented bellow. Note that, besides the declaration and initialisation of the variables `f` and `i` happen simultaneously in the CAO code, they are done separately in the EasyCrypt procedure. This code structure helps when adding assertions to reason about the safety of the expressions that are being assigned to the variables.

```
module Factorial = {

  proc fact (n : CAO_int) : CAO_int = {

    var f : CAO_int;
    var i : CAO_int;

    f = 1;
    i = 1;

    while (i ≤n) {
      f = Int.( ∗ ) f i;
      i = Int.( + ) i 1;
    }

    return (f);
  }
}.
```

Having the EasyCrypt specification of the CAO function, the CAOVerif tool will then use the contract of the CAO function to generate a Hoare triple lemma, that will be used to attest the functional correctness of the function, as follows. Using the Hoare triple syntax, this lemma can be seen as $\forall n' : CAO\_int, \{n = n' \wedge n' \geq 0\}\ Factorial.fact\ \{res = fact\ n'\}$.

```
lemma fact_contract_proof (n' : CAO_int):
  hoare [Factorial.fact : n = n' ∧n' ≥0 ==> res = lfact n'].
```

The proof script will then be generated according to the commands of the `fact` function in reverse order (recall that EasyCrypt tactics perform a bottom-up analysis). The proof script would be the following.

```
lemma fact_contract_proof (n' : CAO_int):
```

```
  hoare [Factorial.fact : n = n' ∧n' ≥0 ==> res = lfact n'].
proof.
  proc ⇒ //.
  while (1≤i≤n+1 ∧f = lfact(i−1)).
    wp.
    skip.
    smt.
  wp.
  skip.
  smt.
qed.
```

The safety-sensitive EasyCrypt module containing the security assertions is defined in the *factorial_safe.ec* file. This file is imported to the original (*factorial.ec*) file in order to produce a safety proof. The safety-sensitive function is similar to the original, except that every statement is annotated with a safety condition.

```
module Factorial_safe = {

  proc fact (n : CAO_int) : CAO_int = {

    var f : CAO_int;
    var i : CAO_int;

    assert (safe (1));
    f = 1;

    assert (safe (1));
    i = 1;

    assert (safe (i ≤n));
    while (i ≤n) {
      assert (safe (Int.( * ) f i));
      f = Int.( * ) f i;

      assert (safe (Int.( + ) i 1));
      i = Int.( + ) i 1;
    }

    assert (safe (f));
    return (f);
  }
```

```
}.
```

Finally, the execution of CAOVerif ends with the generation of the safety proof. Recalling Section 6.6, the proof will consist of a comparison between the *safety-annotated* function and the *clean* one, through a relational Hoare triple. If both functions are equivalent, then the CAO program is safe. The safety proof and corresponding proof script can be found bellow.

```
equiv fact_safe (n' : CAO_int):
  Factorial.fact ~ Factorial_safe.fact :
  ={n} ∧n{1} = n' ∧n' ≥0 ⇛ ={res} ∧res{1} = Ifact n'.
proof.
  proc ⇒ //.
  while (={i} ∧={f} ∧1≤i{1}≤n{1}+1 ∧f{1} = Ifact(i{1}1)).
    wp.
    skip.
    smt.
  wp.
  skip.
  smt.
qed.
```

In this particular case, both scripts would be equal because the **wp** tactic will consume the assignments and the assertions with just one invocation.

7

CONCLUSIONS AND FUTURE WORK

Domain specific languages (DSL) are a very interesting set of programming languages: by having some well defined and restricted application, they offer a programmer the necessary abstractions and mechanisms to write programs of some complicated domain in a very elegant way. Usually, the code of these languages is not executable and the toolset around the language provides code extraction to common programming languages like C or Java. Cryptographic domain specific languages are of extreme interest because the development of cryptographic software involve knowledge in many different areas and need to be almost computationally invisible.

The code extraction feature, even if proven sound, may not produce valid code because the program written in the domain specific language may contain some error that is undetectable due to the lack of code execution. In cryptography, an error such like this can be catastrophic because it can compromise the security assets of security infrastructure. Code verification is, most of the times, carried out in the produced code, owning that the language of the this code usually has some verification mechanism platform. This fact turns the verification process very complicated because there is no natural way of dealing with data structures specific to the domain of the DSL defined in the generated code language.

In this project we bring these two worlds together: we present a verification platform for CAO- a cryptographic domain specific language - based on a verification toolset specific to the domain of cryptography, EasyCrypt. We extended the CAO language with additional features that turn the process of specifying cryptographic primitives even more natural and re-invent the CAOVerif tool, providing it with a new backend tool - EasyCrypt-, that is more suitable to the domain of cryptography.

We provide an interesting connection between the CAO language and the EasyCrypt platform. With the extensions made to CAO, the language became a natural interface for EasyCrypt. A CAO program can now be seen as a parametrisable implementation and generic specifications, meaning that CAO can now also be used as a specification language in which to express directly the constructions described in cryptographic standards and scientific articles, matching the design principles of EasyCrypt. Therefore, some user to write cryptographic code in a very friendly language and then translate it to an EasyCrypt specification that can be used to reason about it, thus increasing the use cases of the CAO toolchain. This feature reduces the need for some user to learn the specification details of EasyCrypt, allowing him to focus on the proving part of the script.

7.1 OLD CAOVERIF VS. NEW CAOVERIF

An important subject of this work was to compare the two versions CAOVerif and to evaluate the advantages and disadvantages of using EasyCrypt or the Frama-C/Jessie toolset as backend for the tool.

The first important difference resides on the nature of EasyCrypt and Frama-C. The first is an interactive proof assistant, where proofs are developed step-by-step with a very low degree of automation. The second, combined with the Jessie plugin, delivers an interface for many external provers (both SMT solvers and interactive proof assistants) where high degrees of automation can be achieved.

By relying on Frama-C with the Jessie plugin, we are able to present a intuitive graphical interface where, even an inexperienced user, is able to reason about programs using simple mechanisms. It may be the case where some proof goals will not be discharged automatically, which may lead to the need of using an interactive proof assistant like COQ. EasyCrypt does not provide a graphical interface and is not as user friendly. Even though much of the script is generated by our new version of CAOVerif, it may require some knowledge of the toolset and of program logics to potentialise the usage of CAOVerif.

When trying to prove simple properties about programs (like safety properties over integers or arrays), the old CAOVerif tool provides a faster mechanism to deal with these proof goals, since many of them are proved with a simple call to an SMT solver. Nevertheless, since we automatically generate proof scripts to try to prove properties about programs, an user would simply need to run the script and check if all the proof obligations are discharged. In what concerns more difficult properties (like safety properties over inhabitants of some extension field type), it really takes no advantage in using the old version of CAOVerif, since one will not be able to proof the generated proof goals without the use of an interactive proof assistant. One more time, the new version of CAOVerif generates the necessary proof scripts, which may be an advantage when comparing to the older version.

There is one big advantage in using EasyCrypt as backend for the CAOVerif tool: since EasyCrypt is specific to the domain of cryptography, it suits better for a deductive verification tool for a cryptographic domain specific language. By generating an EasyCrypt script from a CAO implementation, we are not only able to reason about functional correctness and safety properties, but also to reason about the security of some scheme implemented in CAO. This is hard if CAOVerif relies on the Frama-C/Jessie toolchain, because SMT solvers will not be able to prove security assumptions over programs and, even if one generates a COQ script of the implementation, one would need to use external libraries to prove security properties of CAO programs in a platform not developed with that aim.

Finally, the Jessie plugin is currently deprecated and it is being replaced by the WP plugin. This means that possible bugs on the Jessie plugin would not be fixed, which would compromise the soundness of CAOVerif. In contrast, EasyCrypt is a recent platform, that is supported by an active team, which grants the users a higher degree of confidence.

7.2 FUTURE WORK

From this work, we propose some main directions for future work:

- With the introduction of the sampling operator in CAO, on is able to define probabilistic programs in the language but one is not able to reason about probabilistic programs using CAO-SL. The introduction of probabilistic annotations in CAO-SL would augment the set of programs that can be analysed by our tool.
- The CAO language may also be improved, not only with features that will approximate the language to the standards, but also with features that would turn the specification of programs a less painful process. The inclusion of more high order features, like *maps* or *folds*, in CAO would contribute to the deliver of better and more user-friendly language. Note that these features are already present in the CALF language (editor).
- One could also extend the CAO-SL language with security reasoning mechanisms. For example, an user could annotate the program with some postcondition $isCPA$ that would mean that the scheme written in CAO is secure under the chosen plaintext attack (CPA) assumption. There are no automatic security proofs, however, our tool could generate all the generic lemmas necessary to prove its security, leaving the proof scripts empty.
- Our tool can be included in a bigger platform with the aim to generate correct-by-construction C programs. Intuitively, one would specify some program in CAO, use CAOVerif to reason about it and then use the CAO compiler to generate C implementations from a CAO program that was verified using CAOVerif.

BIBLIOGRAPHY

José Bacelar Almeida, Manuel Barbosa, Jean-Christophe Filliâtre, Jorge Sousa Pinto, and Bárbara Vieira. Caoverif: An open-source deductive verification platform for cryptographic software implementations. *Sci. Comput. Program.*, 91:216–233, 2014. doi: 10.1016/j.scico.2012.09.019. URL http://dx.doi.org/10.1016/j.scico.2012.09.019.

José Bacelar Almeida, Maria João Frade, Jorge Sousa Pinto, and Simão Melo de Sousa. *Rigorous software development : an introduction to program verification*. Undergraduate topics in computer science. Springer, London, 2011. ISBN 978-0-85729-017-5. URL http://opac.inria.fr/record=b1132575.

Manuel Barbosa, Andrew Moss, Dan Page, Nuno F. Rodrigues, and Paulo F. Silva. Type checking cryptography implementations. In Farhad Arbab and Marjan Sirjani, editors, *Fundamentals of Software Engineering*, volume 7141 of *Lecture Notes in Computer Science*, pages 316–334. Springer Berlin Heidelberg, 2012. ISBN 978-3-642-29319-1. doi: 10.1007/978-3-642-29320-7_21. URL http://dx.doi.org/10.1007/978-3-642-29320-7_21.

Gilles Barthe, Benjamin Grégoire, and Santiago Zanella Béguelin. Formal certification of code-based cryptographic proofs. *SIGPLAN Not.*, 44(1):90–101, January 2009. ISSN 0362-1340. doi: 10.1145/1594834.1480894. URL http://doi.acm.org/10.1145/1594834.1480894.

Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, and Santiago Zanella Béguelin. Computer-aided security proofs for the working cryptographer. In *Proceedings of the 31st Annual Conference on Advances in Cryptology*, CRYPTO'11, pages 71–90, Berlin, Heidelberg, 2011a. Springer-Verlag. ISBN 978-3-642-22791-2. URL http://dl.acm.org/citation.cfm?id=2033036.2033043.

Gilles Barthe, Benjamin Grégoire, Yassine Lakhnech, and Santiago Zanella Béguelin. Beyond provable security verifiable ind-cca security of oaep. In *Proceedings of the 11th International Conference on Topics in Cryptology: CT-RSA 2011*, CT-RSA'11, pages 180–196, Berlin, Heidelberg, 2011b. Springer-Verlag. ISBN 978-3-642-19073-5. URL http://dl.acm.org/citation.cfm?id=1964621.1964640.

Gilles Barthe, Juan Manuel Crespo, Benjamin Grégoire, César Kunz, Yassine Lakhnech, Benedikt Schmidt, and Santiago Zanella-Béguelin. Fully automated analysis of padding-based encryption in the computational model. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, pages 1247–1260, New York, NY, USA, 2013. ACM.

ISBN 978-1-4503-2477-9. doi: 10.1145/2508859.2516663. URL http://doi.acm.org/10.1145/2508859.2516663.

Patrick Baudin, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. Acsl: Ansi/iso c specfication language. Technical report, CEA LIST and INRIA, 2010.

Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *Proceedings of the 1st ACM Conference on Computer and Communications Security*, CCS '93, pages 62–73, New York, NY, USA, 1993. ACM. ISBN 0-89791-629-8. doi: 10.1145/168588.168596. URL http://doi.acm.org/10.1145/168588.168596.

Mihir Bellare and Phillip Rogaway. Optimal asymmetric encryption. In *Advances in Cryptology - EUROCRYPT '94, Workshop on the Theory and Application of Cryptographic Techniques, Perugia, Italy, May 9-12, 1994, Proceedings*, pages 92–111, 1994. doi: 10.1007/BFb0053428. URL http://dx.doi.org/10.1007/BFb0053428.

Mihir Bellare and Phillip Rogaway. Code-based game-playing proofs and the security of triple encryption. Cryptology ePrint Archive, Report 2004/331, 2004. http://eprint.iacr.org/.

Nick Benton. Simple relational correctness proofs for static analyses and program transformations. *SIGPLAN Not.*, 39(1):14–25, January 2004. ISSN 0362-1340. doi: 10.1145/982962.964003. URL http://doi.acm.org/10.1145/982962.964003.

Bruno Blanchet. A computationally sound automatic prover for cryptographic protocols. In *Workshop on the link between formal and computational models*, Paris, France, June 2005.

François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. Why3: Shepherd Your Herd of Provers. In *Boogie 2011: First International Workshop on Intermediate Verification Languages*, pages 53–64, Wroclaw, Poland, 2011. URL https://hal.inria.fr/hal-00790310.

Luca Cardelli. Type systems. In Allen B. Tucker, editor, *The Computer Science and Engineering Handbook*, chapter 103, pages 2208–2236. CRC Press, 1997.

J.I. den Hartog and E.P. de Vink. Verifying probabilistic programs using a hoare like logic. *International journal of foundations of computer science*, 13(3):315–340, 2002. URL http://doc.utwente.nl/55799/.

Edsger Wybe Dijkstra. *A Discipline of Programming*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 1997. ISBN 013215871X.

Manuel Bernardo Barbosa (editor). CACE Deliverable D5.2: formal specification language definitions and security policy extensions, 2009. Available from http://www.cace-project.eu.

Levent Erkök and John Matthews. High assurance programming in cryptol. In *Proceedings of the 5th Annual Workshop on Cyber Security and Information Intelligence Research: Cyber Security and Information Intelligence Challenges and Strategies*, CSIIRW '09, pages 60:1–60:2, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-518-5. doi: 10.1145/1558607.1558676. URL http://doi.acm.org/10.1145/1558607.1558676.

Jean-Christophe Filliâtre and Claude Marché. The why/krakatoa/caduceus platform for deductive program verification. In Werner Damm and Holger Hermanns, editors, *Computer Aided Verification*, volume 4590 of *Lecture Notes in Computer Science*, pages 173–177. Springer Berlin Heidelberg, 2007. ISBN 978-3-540-73367-6. doi: 10.1007/978-3-540-73368-3_21. URL http://dx.doi.org/10.1007/978-3-540-73368-3_21.

Jean-Christophe Filliâtre and Andrei Paskevich. Why3 — where programs meet provers. In Matthias Felleisen and Philippa Gardner, editors, *Programming Languages and Systems*, volume 7792 of *Lecture Notes in Computer Science*, pages 125–128. Springer Berlin Heidelberg, 2013. ISBN 978-3-642-37035-9. doi: 10.1007/978-3-642-37036-6_8. URL http://dx.doi.org/10.1007/978-3-642-37036-6_8.

Jean-Christophe Filliâtre, Léon Gondelman, and Andrei Paskevich. The spirit of ghost code. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification*, volume 8559 of *Lecture Notes in Computer Science*, pages 1–16. Springer International Publishing, 2014. ISBN 978-3-319-08866-2. doi: 10.1007/978-3-319-08867-9_1. URL http://dx.doi.org/10.1007/978-3-319-08867-9_1.

Shafi Goldwasser and Silvio Micali. Probabilistic encryption. *Journal of Computer and System Sciences*, 28(2):270 – 299, 1984. ISSN 0022-0000. doi: http://dx.doi.org/10.1016/0022-0000(84)90070-9. URL http://www.sciencedirect.com/science/article/pii/0022000084900709.

Shai Halevi. A plausible approach to computer-aided cryptographic proofs. Cryptology ePrint Archive, Report 2005/181, 2005. http://eprint.iacr.org/.

C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969. ISSN 0001-0782. doi: 10.1145/363235.363259. URL http://doi.acm.org/10.1145/363235.363259.

The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2004. URL http://coq.inria.fr. Version 8.0.

The EasyCrypt development team. *EasyCrypt Reference Manual*, 2015. URL http://www.easycrypt.info. Version 1.x.

Christine Paulin-Mohring. Introduction to the Calculus of Inductive Constructions. November 2014. URL https://hal.inria.fr/hal-01094195.

Victor Shoup. Sequences of games: a tool for taming complexity in security proofs. Cryptology ePrint Archive, Report 2004/332, 2004. http://eprint.iacr.org/.

Nikhil Swamy, Juan Chen, Cedric Fournet, Pierre-Yves Strub, Karthikeyan Bharagavan, and Jean Yang. Secure distributed programming with value-dependent types. Technical Report MSR-TR-2011-37, March 2011. URL http://research.microsoft.com/apps/pubs/default.aspx?id=141708. This is an extended version of the conference paper (ICFP '11) with the same title. A final version of this full technical report is forthcoming.