Formal Verification of Cryptographic Software Implementations

Bárbara Isabel de Sousa Vieira

UMinho | 2012

Universidade do Minho
Escola de Engenharia

Bárbara Isabel de Sousa Vieira

Formal Verification of Cryptographic Software Implementations

June,2012

Universidade do Minho
Escola de Engenharia

Bárbara Isabel de Sousa Vieira

Formal Verification of Cryptographic Software Implementations

Dissertation for the Ph.D. degree in Informatics

Supervisor:
Professor Doutor Manuel Bernardo Barbosa

June, 2012

Acknowledgements

My journey as a PhD student finally came to the end, after almost three years and a half. In the beginning I was not really aware of what doing a PhD is, and I was really doubting if I should or not embark on this long trip. I think I felt inspired by the words of my supervisor, Manuel Barbosa, who patiently explained me what is adopting research as a lifestyle. For those words, I really want to say: thank you! You were completely right! I really love what I'm doing now! Somewhat this defines me, this is who I am.

These last years looked like a roller coaster of emotions, with many many ups and downs, and go through them was not possible without the support and love of very important people. To these people I want to dedicate a few words.

First of all I want to thank my supervisor Manuel Barbosa, for his supportive words in the most difficult times and guiding me during these years. Thanks for all (sometimes crazy) fruitful discussions and for unconditionally supporting all my decisions such as having different research experiences in other centers out of Portugal. Besides, you were always there! Many many thanks.

I was really a lucky girl, who had not one, but three supervisors! The wisdom, incisive comments, patience and brilliant mind of José Bacelar were fundamental for the good results of this work. Also the tranquility, understanding, wise words in the crucial moments and expertise of Jorge Sousa Pinto underpinned all the good results achieved. For them, goes a huge thank you.

Of course, I will never forget the time I spent at INRIA Saclay Île-de-France on the supervision of Jean-Christophe Filliâtre. I really loved the three months internship in the first semester of 2009! Jean-Christophe Filliâtre is really an inspiring and motivating person and I really loved to work with him.

I want to say thanks to many more people, starting by my office colleagues of always, Miguel Marques (who unfortunately, to my regret, left the office a year ago), Hugo Pacheco (the funny crabber) and Hugo Macedo (the political guy). To my new and

recent colleagues Paulo Silva, Jácome Cunha and Nuno Macedo I want to also say thanks. Of course I cannot forget the crazy teacher with whom I really like to talk and laugh, José Bernardos Barros (aka JBB).

I would like to thank all my outside work friends. Again in no special oder: Andreia Machado, Hugo Machado, Miguel Domingues, Ana Pereira, Bruno Marques, Maria Joana, João Pedro Silva, Sofia Pontes and Marilia Braga. To all of you, thank you very much for everything.

Finally, to the most important persons of my life, my love Marco and my mother, I really want to say thank you. You are the persons that I love most and without you I could not have done this. Mom thank you for always believing in me and make me believe in me too. Marco thank you for supporting me, trust and encourage me. Yes, I will marry you (some day). You are the one!

The work in this thesis has been carried out at the Departamento de Informática, Universidade do Minho, Portugal. The author was funded by the CACE European project (co-financed by the European Commission under EU Framework Programme 7), by the research center INRIA Saclay-Île-France and by Fundação para a Ciência e Tecnologia (Portugal), under grant number SFRH / BD / 75010 / 2010.

Formal Verification of Cryptographic Software Implementations

Abstract

Security is notoriously difficult 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. This thesis is organized in two parts.

The first part is focused on the identification of relevant security policies that may be at play in cryptographic systems, as well as the language-based mechanisms that can be used to enforce such policies in those systems. We propose methodologies based on deductive verification to formalise and verify relevant security policies in cryptographic software. We also show the applicability of those methodologies by presenting some case studies using a deductive verification tool integrated in the Frama-c framework.

In the second part we propose a deductive verification tool (CAOVerif) for a domain-specific language for cryptographic implementations (CAO). Our aim is to apply the methodologies proposed in the first part of this thesis work to verify the cryptographic implementations written in CAO. The design of CAOVerif follows the same approach used in other scenarios for general-propose languages and it is build on top of a plug-in from the Frama-c framework. At the very end, we conclude the work of this thesis by reasoning about the soundness of our verification tool.

Verificação Formal de Implementações de Software Criptográfico

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. O trabalho desta tese está organizado em duas partes.

A primeira parte foca-se essencialmente na identificação de políticas de segurança relevantes nos sistemas criptográficos, bem como nos mecanismos baseados em linguagens que podem ser aplicados para garantir tais políticas. Neste contexto, propomos metodologias baseadas em verificação dedutiva para formalizar e verificar políticas de segurança. Mostramos também como essas metodologias podem ser aplicadas na verificação de casos de estudo reais, utilizando a ferramenta de verificação dedutiva integrada na ferramenta Frama-c.

Na segunda parte, propomos uma ferramenta de verificação dedutiva (CAOVerif) para uma linguagem de domínio específico para implementações criptográficas (CAO). O desenvolvimento de tal ferramenta tem como objectivo aplicar as metodologias desenvolvidas na primeira parte deste trabalho às implementações criptográficas definidas em CAO. O desenho desta ferramenta segue a mesma aproximação de outras ferramentas de verificação dedutiva já existentes para outras linguagens. Concluímos o trabalho desenvolvido dando um prova formal da correcção da ferramenta.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Context

Software implementations of cryptographic algorithms and protocols are at the core of security functionality in many IT products. However, the development of this class of software products is understudied as a domain-specific niche in software engineering.

The development of cryptographic software is clearly distinct from other areas of software engineering due to a combination of factors. Firstly, cryptography is an inherently interdisciplinary subject. The design and implementation of cryptographic software draws on skills from mathematics, computer science and electrical engineering. The assumption that such a rich body of research can be absorbed and applied without error is tenuous for even the most expert software engineer. Secondly, security is notoriously difficult to sell as a feature in software products, even when clear risks such as identity theft and fraud are evident. An important implication of this fact is that security needs to be as close to invisible as possible in terms of computational and communication load. As a result, it is critical that cryptographic software be optimised aggressively, without altering the security semantics. Finally, typical software engineers develop systems focused on desktop class processors within computers in our offices and homes. The special case of cryptographic software is implemented on a much wider range of devices, from embedded processors with very limited computational power, memory and autonomy, to high-end servers, which demand high-performance and low-latency. Not only must cryptographic software engineers understand each platform and the related security requirements, they must also optimise each algorithm with respect to each platform, since each one will have vastly different performance characteristics.

CACE (Computer Aided Cryptography Engineering[1]) was an European Project that aimed to improve on the lack of support currently offered to cryptographic software engineers. The main output of the project was the development of a tool-box of domain-specific languages, compilers and libraries, that support the production of high quality cryptographic software. Specific components within the produced tool-box address particular software development problems and processes; and combined use of the constituent tools is enabled by designed integration between their interfaces. In particular, one of the work packages in the project focused on adding formal methods technology to the CACE tool-box, as a means to increase the degree of assurance than can be provided by the development process. The adopted strategy was to adapt existing results in the state of the art of formal methods to the specific domain of cryptography. This thesis is a direct result of a specific challenge put forward within this CACE work package: to explore the application of deductive program verification techniques to implementations of low-level cryptographic primitives, and extend the CACE toolbox to permit carrying out this type of formal verification over implementations of low-level cryptographic primitives in a domain-specific language called CAO.

In the remainder of this introductory chapter, we will give an overview of what this challenge entails and how it was approached, highlight the main results in this work, and explain how their presentation is organized throughout this thesis.

1.2 The CAO language

The CAO language aims to allow 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, so as to eliminate some of the difficulties arising in the use of languages such as C or Java. However, unlike the languages used in mathematical software packages such as Magma[2] or Maple[3], 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

[1]http://www.cace-project.eu
[2]http://magma.maths.usyd.edu.au/magma
[3]http://www.maplesoft.com

finite field arithmetic for Elliptic Curve Cryptography (ECC).

CAO preserves some higher-level features to be familiar to an imperative programmer, with a syntax that is very close to that of C, 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 permit 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. 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. 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.

As mentioned above, the driving objective of this work was to develop a tool that allows formally verifying CAO implementations of cryptographic algorithms. Such a tool should carry over to the specific domain of cryptography the capabilities of state-of-the-art deductive verification tools.

A necessary stepping stone towards achieving this goal was to evaluate the capabilities of existing deductive verification platforms, and to explore their applicability to the formal verification of cryptographic software.

1.3 Deductive program verification

Program verification refers to obtaining assurance that a particular piece of software will display a particular property when it is executed. Program verification can be performed using different methodologies, with more or less support from development tools, and providing varying degrees of assurance.

One can envision tool support for *checking* whether a particular property is present, and also tool support for *enforcing* a particular property. Verification can also be dynamic and/or static. In the first case, verification is performed at run-time, and it can be implemented by using reference monitors. These mechanisms detect a problem, whenever they are about to occur, and take appropriate mitigation action. Static verification is carried out before the code is executed and has obvious performance and reliability advantages.

Another important aspect of verification is whether the outputs of the verification process, i.e. the assurance obtained, must be transferred to a party other than the software developer. In many scenarios this is in fact the case. Such guarantees may need to be transferred to the consumer of the product, for example when a company is outsourcing part of the development of an application and wants assurance that the supplier is meeting the requirements. One may also need to transfer assurance to a certification body, for example when a company is trying to get a product approved for a particular use. Certification standards such as the Common Criteria [31, 32, 33] point to formal methods as a technology that enables the development of verification methods which can be used to transfer assurance.

This work focuses on technology that can be used in these demanding scenarios. The techniques employed allow formally proving properties of programs statically. These properties are established through deductive reasoning based on Hoare Logic [54, 47], and the verification process is therefore termed *deductive verification*. These techniques are brought to practice through the use of *contracts* – specifications consisting of preconditions and post-conditions, annotated into the programs. In recent years verification tools based on contracts have become more and more popular, as their scope evolved from toy languages to very realistic fragments of languages like C, C#, or Java [30, 25, 44, 14, 16, 46].

In a nutshell, the typical architecture of a deductive verification infrastructure consists of a verification condition generator (VCGen for short) and a proof tool, which may be either an automatic theorem prover or an interactive proof assistant. The VCGen reads in the annotated code (which contains contracts and other annotations meant to facilitate the verification, such as loop invariants and variants) and produces a set of proof obligations known as *verification conditions*, that are sent to the proof tool. The correctness of the VCGen guarantees that if all the proof obligations are valid then the program is correct with respect to its specification.

In this work we have built upon an existing verification platform called `Frama-c` [21]. This is a framework for the static analysis of `C` programs annotated using the ANSI-C

Specification Language (ACSL [21]). Frama-c includes a deductive verification tool called Jessie, which contains a multi-prover VCGen [45], that can target a set of proof tools including the Coq proof assistant [89], and the Simplify [39] and Alt-Ergo [34] automatic theorem provers. Frama-c has proven to be a good choice for two main reasons. Firstly, it includes a powerful deductive verification tool for C programs, which is by far the programming language that shares the most characteristics with CAO, and for which there exists huge volume of cryptographic implementations that can be used as case studies. Secondly, the Frama-c architecture is highly modular, which allowed us to reuse some of its underlying components in the implementation of the CAO deductive verification tool.

The first aspect facilitated our exploration of the potential applications of deductive verification over cryptographic software. This led to interesting results that, not only guided us in the design a new deductive verification tool for CAO – called CAO-Verif – but are also independent contributions in the area of formal verification. The second possibility was critical in the management of our development work and greatly simplified carrying over the applicability of the previous results to the CAO language.

1.4 Main results

The results in this thesis are presented in two parts. Part I focuses on our results on applying deductive verification techniques to cryptographic software. Part II presents our achievements in the development of a deductive verification tool for the CAO domain specific language.

Part I - Deductive verification of cryptographic software

Extensive work was done in the CACE project [65] to identify and classify a set of relevant properties of cryptographic software implementations in a wide range of application scenarios. Given the security-critical nature of cryptographic software, we refer to these properties as *security properties*.

In this part of the thesis we focus on a subset of properties that are applicable to C or CAO implementations of low-level cryptographic primitives. The results we present here stem from our effort in determining how these properties can be formalized and verified using deductive verification techniques, with subsequent validation of our approach using real-world examples of C cryptographic code and Frama-c. The security properties we address are diverse and include safety properties, where the goal is to exclude fatal errors and exception conditions such as those arising from invalid memory accesses and numeric errors; functional correctness properties where the aim is to establish that the input/output behavior of a program agrees with its specification; and information flow properties, where the goal is to ensure compliance to security policies that exclude sensitive information leakage or contamination.

Technically, we extend the notion of self-composition introduced by Barthe et al. [17] to a generalized notion of *composition-based proofs* and apply it to these new scenarios. We show that it can be used to tackle interesting new properties including the correctness of code optimizations that are common in C cryptographic code; non-interference properties such as the absence of error propagation on stream ciphers; and adherence to state-of-the-art side-channel attacks countermeasures that are adopted in the C implementation of the NaCl (read *salt*) cryptographic library[4] developed in the CACE project.

We are also concerned with the level of automation that can be introduced to help the end-user in the application of our techniques. One of the main hurdles in scaling deductive verification techniques to more complex verification targets is perhaps the need for intensive use intervention in annotating the code and managing the discharge of a large number of verification conditions. This is also true with our techniques, and hence we have explored the use of *natural invariants* (a technique to derive loop invariants and detailed in Chapter 4) as a means to increase the level of automation and the complexity of the use cases that can be tackled using composition based proofs and deductive verification platforms.

The main results comprised in this part of the thesis were published in [4, 6, 7] and in [5].

Part II - A deductive verification tool for CAO

The results above show that a tool such as Frama-c has great potential for verifying a wide variety of security-relevant properties in cryptographic software implementations. However, our experience also showed us that the intrinsic characteristics of the C language make it a hard target for formal verification, particularly when the goal is to increase the level of automation. This problem is amplified when the verification target is in the domain of cryptography, because implementations typically explore language

[4]`http://nacl.cr.yp.to`

constructions that are little used in other application areas, including bit-wise operations, unorthodox control-flow (loop unrolling, single-iteration loops, break statements, etc.), intensive use of macros, etc.

In constructing a deductive verification tool for CAO we therefore had the opportunity to take advantage of the characteristics of this programming language to construct a domain-specific verification tool, allowing for the same generic verification techniques that can be applied over C implementations, simplifying the verification of security-relevant properties, and hopefully providing a higher degree of automation.

In the second part of the thesis we describe the design and implementation of CAO-Verif (the deductive verification tool for CAO). We show that CAO presents interesting challenges for formal verification, concerning not only its rich type system, but also the cryptography-oriented language constructions that it offers. We describe how we tackle these problems, namely by presenting what we believe is the first formalisation in first-order logic of the rich mathematical data types that are used in cryptography in the context of deductive verification. We also demonstrate that the development time of such a complex verification tool can be greatly reduced by relying on the Jessie plug-in of the Frama-c framework as a back-end.

We base our presentation on real-world examples of CAO code, extracted from the open-source code of the NaCl cryptography library. We show how we fine tuned our tool to enable the fully automatic formal verification of simple properties (in particular safety properties), and also how more ambitious proof goals (arising in general proofs of functional correctness) can be factored by resorting to specific lemmas (that will be introduced in Chapter 4).

We complement this work by giving a formal proof of the soundness of our verification tool. Part of the results presented in the second part of the thesis have been published in [11].

1.5 Thesis organization

Chapter 2 This chapter introduces the main theoretical concepts underlying the work presented in this thesis. More specifically, it describes two toy languages which capture the main features of CAO and Jessie, as well as the formalisms we have adopted to tackle the security properties addressed in this thesis.

Chapter 3 In this chapter we revise related work in the study of formal techniques that can be used to verify general propose software systems, specially focusing on deductive verification techniques. We also give a broader view of the Frama-c framework and its deductive verification plug-in, named Jessie, as well as, of other formal approaches commonly used to verify the security properties addressed in our work.

Part I - Deductive verification of cryptographic software

Chapter 4 This chapter focuses on the formalisation and verification of a specific set of functional correctness and information flow properties in cryptographic software. We apply deductive verification techniques to formally verify safety; correctness with respect to reference implementations and absence of error propagation properties. The main results comprised in this chapter were published in [4, 6] and in [7].

Chapter 5 In this chapter we extend the work presented in the previous chapter to demonstrate the power of our methodology to reduce exposure to *side-channel attacks*. We focus our attention on the recently proposed NaCl cryptographic library and we analyse a set of high-level non-functional security properties, whose purpose is to minimise exposure to side-channel attacks in the C implementation of this library. We introduce an instrumented trace semantics which plays a key role here, since it makes possible to express these policies as non-interference-like properties, thus allowing us to bridge an important gap between the general, theoretical formulation of security properties employed in our previous work, and the real-world concerns and coding practices of cryptographers. The main results comprised in this chapter were published in [5].

Part II - CAOVerif: A deductive verification tool for CAO

Chapter 6 In this chapter we describe the design and implementation details of the deductive verification tool for CAO, named CAOVerif. We present the formalisation in first-order logic of the rich mathematical data types of this language and also demonstrate that the development time of such a complex verification tool can be greatly reduced by relying on the Jessie plug-in of the Frama-c framework as a back-end. The main results comprised in this chapter were published in [11].

Chapter 7 In this chapter we establish the correctness of our approach by giving a formal proof of the soundness of the CAOVerif tool, i.e., that it only allows to prove the correctness of programs that are indeed correct.

Chapter 8 This chapter summarises the work presented in this thesis, by giving the main conclusions and presenting directions for future work.

Chapter 2

Preliminaries

In this chapter we introduce some important notions and definitions that will be used throughout this thesis. We begin with the description of two simple imperative languages which we use to formalise the security properties we have addressed; and we conclude the chapter by presenting the formal definition (we have adopted) of such properties.

2.1 *While*^{C} language

While^C is a simple imperative language with integer expressions, bounded arrays and booleans. This is an abstraction of a higher-level language such as CAO, where we assume that the type information for array variables includes length information (these and other features of the CAO language are detailed in Chapter 6). For this reason, when formalising the operational semantics of the language, we consider the existence of the function len, which given an array retrieves its length. The syntax of the language is shown in Figure 2.1.

Integer literals are ranged by **n** and boolean literals are **true** and **false**. Integer and boolean variables are ranged by $\mathtt{x}^i$ and $\mathtt{x}^b$, respectively, and array variables are ranged by a. For simplicity, whenever it is implicit from the context, we will omit the superscripts i and b when referring to integer and boolean expressions. We will also write e to refer to expressions in general.

Integers	$\mathsf{Int} \ni e^i$	$::=$	$\mathbf{n} \mid \mathrm{x}^i \mid \mathtt{a[}e\mathtt{]} \mid e^i + e^i \mid e^i - e^i \mid e^i * e^i \mid e^i \;/\; e^i$
Booleans	$\mathsf{Bool} \ni e^b$	$::=$	$\mathbf{true} \mid \mathbf{false} \mid \mathrm{x}^b \mid e^b == e^b \mid e^b \;!= e^b \mid e^i < e^i \mid e^i > e^i$
Commands	$\mathsf{Comm} \ni C$	$::=$	$\mathbf{skip} \mid \mathtt{x} := e \mid \mathtt{a[}e^i\mathtt{]} := e^i \mid \mathbf{if}\ (e^b)\ \{C_1\}\ \mathbf{else}\ \{C_2\} \mid$
			$\mathbf{while}\ (e^b)\ C \mid C_1\ ;\ \ C_2$

Figure 2.1: Syntax of *While*^{C}

Semantics We consider an error semantics for *While*^{C} to deal with the occurrence of run-time errors. The semantics is extended with a special error value (**error**) used to denote the result of undefined expressions. Essentially, this error value is used to capture divisions by zero and out-of-bounds array accesses (we remark that in a language such as CAO, the set of runtime errors is much greater). Integer expressions are interpreted as (unbound) integers ($\mathbb{Z}$), boolean expressions as values of $\{\mathbf{true}, \mathbf{false}\}$, and arrays as functions from integers to integers ($\mathbb{Z} \to \mathbb{Z}$). We consider a State type defined as the cartesian product of the corresponding interpretation domains (each variable is associated to a particular position) and ranged by s. We also consider an equivalence relation $\equiv$ that captures equality on states.

The semantics of expressions is given by a functional $[\![.]\!]$ which maps every expression e to a function $[\![e]\!] : \mathsf{State} \to \mathcal{D}$, where $\mathcal{D}$ corresponds to the interpretation domain of e. This functional $[\![.]\!]$ maps every integer expression e^i to a function $[\![e^i]\!] : \mathsf{State} \to (\mathbb{Z} \bigcup \{\mathbf{error}\})$ and every boolean expression e^b to a function $[\![e^b]\!] : \mathsf{State} \to (\{\mathbf{true}, \mathbf{false}\} \bigcup \{\mathbf{error}\})$, as follows:

- $[\![e^i]\!] : \mathsf{State} \to \mathbb{Z} \bigcup \{\mathbf{error}\}$ is inductively defined by:

$$[\![\mathbf{n}]\!](s) = \mathbf{n} \qquad [\![\mathrm{x}^i]\!](s) = s(\mathrm{x}^i)$$

$$[\![e^i_1\ op\ e^i_2]\!](s) = \begin{cases} [\![e^i_1]\!](s)\ op\ [\![e^i_2]\!](s) & \text{if } [\![e^i_1]\!](s) \neq \mathbf{error} \wedge [\![e^i_2]\!](s) \neq \mathbf{error} \\ \mathbf{error} & \text{otherwise} \end{cases}$$

$$\text{for } op \in \{+, -, *\}$$

$$[\![e^i_1 \;/\; e^i_2]\!](s) = \begin{cases} [\![e^i_1]\!](s) \;/\; [\![e^i_2]\!](s) & \text{if } [\![e^i_1]\!](s) \neq \mathbf{error} \ \wedge [\![e^i_2]\!](s) \neq \mathbf{error} \wedge \\ & [\![e^i_2]\!](s) \neq 0 \\ \mathbf{error} & \text{otherwise} \end{cases}$$

$$[\![\mathsf{a}[e^i]]\!](s) = \begin{cases} s(a)([\![e^i]\!](s)) & \text{if } [\![e^i]\!](s) \neq \textbf{error} \wedge 0 \leq [\![e^i]\!](s) < \mathsf{len(a)} \\ \textbf{error} & \text{otherwise} \end{cases}$$

- $[\![e^b]\!]$: $\mathsf{State} \to \{\textbf{true}, \textbf{false}\} \bigcup \{\textbf{error}\}$ is inductively defined by:

$$[\![\textbf{true}]\!](s) = \textbf{true} \qquad [\![\textbf{false}]\!](s) = \textbf{false} \qquad [\![\mathbf{x}^b]\!](s) = s(\mathbf{x}^b)$$

$$[\![e_1 \; op \; e_2]\!](s) = \begin{cases} [\![e_1]\!](s) \; op \; [\![e_2]\!](s) & \text{if } [\![e_1]\!](s) \neq \textbf{error} \wedge [\![e_2]\!](s) \neq \textbf{error} \\ \textbf{error} & \text{otherwise} \end{cases}$$

$$\text{for } op \in \{==, !=\}$$

$$[\![e^i_1 \; op \; e^i_2]\!](s) = \begin{cases} [\![e^i_1]\!](s) \; op \; [\![e^i_2]\!](s) & \text{if } [\![e^i_1]\!](s) \neq \textbf{error} \wedge [\![e^i_2]\!](s) \neq \textbf{error} \\ \textbf{error} & \text{otherwise} \end{cases}$$

$$\text{for } op \in \{<, >\}$$

Note that, in the evaluation semantics of the comparison operators ($op \in \{==, !=\}$), the expressions e_1 and e_2 can be either booleans or integer expressions.

We consider a natural (big-step) semantics for *While*^{C}. The evaluation semantics of commands is given by the relation $\Downarrow \subseteq \texttt{Comm} \times \mathsf{State} \times (\mathsf{State} \bigcup \{\textbf{error}\})$ (where **error** denotes a special error state) inductively defined by the rules given in Figure 2.2. The judgement $(C, s) \Downarrow s'$ means that the evaluation of the program C in the initial state s results in a final state s'. The co-domain of $\Downarrow$ includes an error state, the idea being that if some of the program expressions involved evaluate to an error value, the corresponding program also evaluates to an error state. The function upd: $(\mathbb{Z} \to \mathbb{Z}) \to \mathbb{Z} \to \mathbb{Z} \to (\mathbb{Z} \to \mathbb{Z})$, used in the definition of the semantics rules to update the values of an array, is defined by:

$$\mathsf{upd}(a, i, x) = \lambda j. \begin{cases} a(j) & \text{if } i \neq j \\ x & \text{if } i = j \end{cases}$$

which means that $\mathsf{upd}(\mathsf{a}, i, x)$ is equal to the function that maps an index j to the value x when $i = j$ and to $\mathsf{a}(j)$ when $i \neq j$.

1. $(\textbf{skip}, s) \Downarrow s$
2. If $[\![e]\!](s) = \textbf{error}$ then $(\texttt{x:=}e, s) \Downarrow \textbf{error}$
3. If $[\![e]\!](s) \neq \textbf{error}$ then $(\texttt{x:=}e, s) \Downarrow s[x \leftarrow [\![e]\!](s)]$
4. If $[\![\texttt{a[}e_1\texttt{]}]\!](s) = \textbf{error}$ or $[\![e_2]\!](s) = \textbf{error}$ then $(\texttt{a[}e_1\texttt{]:=}e_2, s) \Downarrow \textbf{error}$
5. If $[\![\texttt{a[}e_1\texttt{]}]\!](s) \neq \textbf{error}$ and $[\![e_2]\!](s) \neq \textbf{error}$ then
   $(\texttt{a[}e_1\texttt{]:=}e_2, s) \Downarrow s[a \leftarrow \mathsf{upd}(s(a), [\![e_1]\!](s), [\![e_2]\!](s))]$
6. If $[\![e]\!](s) = \textbf{error}$ then $(\textbf{if}\ (e)\ \{C_1\}\ \textbf{else}\ \{C_2\}, s) \Downarrow \textbf{error}$
7. If $[\![e]\!](s) = \textbf{true}$ and $(C_1, s) \Downarrow s'$ then $(\textbf{if}\ (e)\ \{C_1\}\ \textbf{else}\ \{C_2\}, s) \Downarrow s'$
8. If $[\![e]\!](s) = \textbf{false}$ and $(C_2, s) \Downarrow s'$ then $(\textbf{if}\ (e)\ \{C_1\}\ \textbf{else}\ \{C_2\}, s) \Downarrow s'$
9. If $[\![e]\!](s) = \textbf{error}$ then $(\textbf{while}\ (e)\ C, s) \Downarrow \textbf{error}$
10. If $[\![e]\!](s) = \textbf{true}$ and $(C, s) \Downarrow s'$ and $(\textbf{while}\ (e)\ C, s') \Downarrow s''$
    then $(\textbf{while}\ (e)\ C, s) \Downarrow s''$
11. If $[\![e]\!](s) = \textbf{false}$ then $(\textbf{while}\ (e)\ C, s) \Downarrow s$
12. If $(C_1, s) \Downarrow \textbf{error}$ then $(C_1; C_2, s) \Downarrow \textbf{error}$
13. If $(C_1, s) \Downarrow s'$ and $(C_2, s') \Downarrow s''$ then $(C_1; C_2, s) \Downarrow s''$

Figure 2.2: Evaluation semantics of *While*^{C} (**error** denotes the error state)

Hoare Logic for *While*^{C}

In this section we introduce the *Hoare logic* formal system [54] to reason about the correctness of *While*^{C} programs. The correctness is expressed in terms of *safety-sensitive Hoare triples* used to specify the desired behaviour of the underlying programs. Recall that we defined an error semantics for the *While*^{C} language to capture possible errors during the execution of programs, therefore, the correctness of *While*^{C} programs is defined accordingly.

Given a command C, a precondition φ and a postcondition ψ, a *safety-sensitive* Hoare triple, denoted by $\{\!|\varphi|\!\}\ C\ \{\!|\psi|\!\}$, has the following meaning: if φ holds in a given state and C is executed in that state and terminates, then that state is different from the **error** state and ψ holds (in that state).

The syntax of these first-order logic predicates (φ and ψ, also called assertions) is given by:

$$\begin{aligned}\phi,\psi ::= &\ \mathbf{true} \mid \mathbf{false} \mid e_1\ ==\ e_2 \mid e_1\ !=\ e_2 \mid e_1^i\ <\ e_2^i \mid e_1^i\ >\ e_2^i \mid \\ &\neg\phi \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \phi_1 \to \phi_2 \mid \phi_1 \leftrightarrow \phi_2 \mid \forall x.\ \phi \mid \exists x.\ \phi \mid\end{aligned}$$

and the semantics is given by the function $[\![\phi]\!]_{\mathcal{M}}$: State $\to \{\mathbf{true}, \mathbf{false}\}$ inductively defined by:

$$\begin{array}{lll}
[\![\mathbf{true}]\!]_{\mathcal{M}}(s) = \mathbf{true} & [\![\mathbf{false}]\!]_{\mathcal{M}}(s) = \mathbf{false} & \\
[\![e_1 == e_2]\!]_{\mathcal{M}}(s) = \mathbf{true} & \text{iff } [\![e_1]\!](s) = [\![e_2]\!](s) & \\
[\![e_1 != e_2]\!]_{\mathcal{M}}(s) = \mathbf{true} & \text{iff } [\![e_1]\!](s) \neq [\![e_2]\!](s) & \\
[\![e_1^i < e_2^i]\!]_{\mathcal{M}}(s) = \mathbf{true} & \text{iff } [\![e_1^i]\!](s) < [\![e_2^i]\!](s) & \\
[\![e_1^i > e_2^i]\!]_{\mathcal{M}}(s) = \mathbf{true} & \text{iff } [\![e_1^i]\!](s) > [\![e_2^i]\!](s) & \\
[\![\psi \wedge \phi]\!]_{\mathcal{M}}(s) = \mathbf{true} & \text{iff } [\![\psi]\!]_{\mathcal{M}}(s) = \mathbf{true} \text{ and } [\![\phi]\!]_{\mathcal{M}}(s) = \mathbf{true} & \\
[\![\psi \vee \phi]\!]_{\mathcal{M}}(s) = \mathbf{true} & \text{iff } [\![\psi]\!]_{\mathcal{M}}(s) = \mathbf{true} \text{ or } [\![\phi]\!]_{\mathcal{M}}(s) = \mathbf{true} & \\
[\![\psi \to \phi]\!]_{\mathcal{M}}(s) = \mathbf{true} & \text{iff } [\![\psi]\!]_{\mathcal{M}}(s) = \mathbf{false} \text{ or } [\![\phi]\!]_{\mathcal{M}}(s) = \mathbf{true} & \\
[\![\psi \leftrightarrow \phi]\!]_{\mathcal{M}}(s) = \mathbf{true} & \text{iff } [\![\psi]\!]_{\mathcal{M}}(s) = \mathbf{true} \text{ and } [\![\phi]\!]_{\mathcal{M}}(s) = \mathbf{true} & \\
 & \text{or } [\![\psi]\!]_{\mathcal{M}}(s) = \mathbf{false} \text{ and } [\![\phi]\!]_{\mathcal{M}}(s) = \mathbf{false} & \\
[\![\forall x.\ \psi]\!]_{\mathcal{M}}(s) = \mathbf{true} & \text{iff } [\![\psi]\!]_{\mathcal{M}}(s[x \leftarrow v]) = \mathbf{true} & \text{for all values } v \\
[\![\exists x.\ \psi]\!]_{\mathcal{M}}(s) = \mathbf{true} & \text{iff } [\![\psi]\!]_{\mathcal{M}}(s[x \leftarrow v]) = \mathbf{true} & \text{for some value } v
\end{array}$$

where $\mathcal{M}$ is the first-order model that interprets expressions/operations according to the semantics of *While*^{C} (partial operations are totalized in some pre-determined way, e.g. divisions by zero return zero and arrays do not overflow). Accordingly, the semantics of *safety-sensitive* Hoare triples is given by the following definition.

Definition 1. *(Safety-sensitive Hoare triple)*

The semantics of a safety-sensitive Hoare triple $\{\!|\varphi|\!\}\ C\ \{\!|\psi|\!\}$ *is given by a function* $[\![\{\!|\varphi|\!\}\ C\ \{\!|\psi|\!\}]\!]_{\mathcal{M}}$: State $\to \{\textbf{\textit{true}}, \textbf{\textit{false}}\}$ *defined as follows:*

$$\begin{aligned}[\![\{\!|\varphi|\!\}\ C\ \{\!|\psi|\!\}]\!]_{\mathcal{M}}(s) = \textbf{\textit{true}} \quad \text{iff}\ \ \text{if } [\![\varphi]\!]_{\mathcal{M}}(s) = \textbf{\textit{true}} \text{ and } (C, s) \Downarrow s' \text{ then} \\ s' \neq \mathbf{error} \text{ and } [\![\psi]\!]_{\mathcal{M}}(s') = \textbf{\textit{true}}\end{aligned}$$

and C is said to be partially correct with respect to the predicate φ *if the triple* $\{\!|\varphi|\!\}\ C\ \{\!|\psi|\!\}$ *is valid.*

$$\text{(skip)}\,\frac{}{\{\!|\varphi|\!\}\ \textbf{skip}\ \{\!|\psi|\!\}}\ \text{if } \varphi \to \psi$$

$$\text{(assign)}\,\frac{}{\{\!|\varphi|\!\}\ x := e\ \{\!|\psi|\!\}}\ \text{if } \varphi \to \mathsf{safe}(e) \land \varphi \to \psi[e/x]$$

$$\text{(assign-array)}\,\frac{}{\{\!|\varphi|\!\}\ \mathsf{a}[e_1] := e_2\ \{\!|\psi|\!\}}\ \text{if } \varphi \to \mathsf{safe}(\mathsf{a}[e_1]) \land \varphi \to \mathsf{safe}(e_2) \land \varphi \to \psi[\mathsf{upd}(\mathsf{a}, e_1, e_2)/\mathsf{a}]$$

$$\text{(if-else)}\,\frac{\{\!|e \land \varphi|\!\}\ C_1\ \{\!|\psi|\!\} \qquad \{\!|\neg e \land \varphi|\!\}\ C_2\ \{\!|\psi|\!\}}{\{\!|\varphi|\!\}\ \textbf{if}\ (e)\ \{C_1\}\ \textbf{else}\ \{C_2\}\ \{\!|\psi|\!\}}\ \text{if } \varphi \to \mathsf{safe}(e)$$

$$\text{(seq)}\,\frac{\{\!|\varphi|\!\}\ C_1\ \{\!|\theta|\!\} \qquad \{\!|\theta|\!\}\ C_2\ \{\!|\psi|\!\}}{\{\!|\varphi|\!\}\ C_1;\ C_2\ \{\!|\psi|\!\}}$$

$$\text{(while)}\,\frac{\{\!|\theta \land \mathsf{safe(e)} \land e|\!\}\ C\ \{\!|\theta \land \mathsf{safe(e)}|\!\}}{\{\!|\varphi|\!\}\ \textbf{while}\ \{\theta\}\ (e)\ \{C\}\ \{\!|\psi|\!\}}\ \text{if } \varphi \to (\theta \land \mathsf{safe}(e)) \land (\theta \land \mathsf{safe(e)} \land \neg e \to \psi)$$

$$\text{(assert)}\,\frac{}{\{\!|\varphi|\!\}\ \textbf{assert}\ \theta\ \{\!|\psi|\!\}}\ \text{if } \varphi \to (\theta \land \psi)$$

Figure 2.3: Inference system for *safety-sensitive* Hoare triples

We define next an axiomatic semantics for the *While*^{C} language with annotations, in the form of an inference system for safety-sensitive Hoare triples, together with the corresponding verification condition generator (VCGen).

Axiomatic semantics We consider now the *While*^{C} language extended with annotations. Annotated *While*^{C} programs include, besides pre- and postconditions, *loop invariants* – predicates which hold during the loop execution; and general *assertions* – predicates inserted between statements that must be valid when the program reaches that point during its execution. The inference system which defines the axiomatic semantics of this *While*^{C} annotated language is presented in Figure 2.3, where $\mathsf{safe}(e)$ is a *safety-condition* whose validity implies that the evaluation of e in any state will not produce an error. This predicate is inductively defined by:

$$\mathsf{safe}(\textbf{true}) = \textbf{true} \qquad \mathsf{safe}(\textbf{false}) = \textbf{true}$$

$$\mathsf{safe}(\mathsf{x}^{\mathsf{i}}) = \textbf{true} \qquad \mathsf{safe}(\mathsf{x}^{\mathsf{b}}) = \textbf{true}$$

$$\mathsf{safe}(\mathsf{a}[e]) = \mathsf{safe(e)} \land 0 \le e < \mathsf{len(a)} \qquad \mathsf{safe}(\mathsf{e}_1\ op\ \mathsf{e}_2) = \mathsf{safe}(\mathsf{e}_1) \land \mathsf{safe}(\mathsf{e}_2)$$

$$\mathsf{safe}(\mathsf{e}_1\ /\ \mathsf{e}_2) = \mathsf{safe}(\mathsf{e}_1) \wedge \mathsf{safe}(\mathsf{e}_2) \wedge e_2 \neq 0 \qquad op \in \{*, +, -, <, >, ==, !=\}$$

The *assign-array* rule refers in its side-condition the upd operator whose semantics is given by $[\![\mathsf{upd}(\mathsf{a}, e_1, e_2)]\!](s) = \mathsf{upd}(s(\mathsf{a}), [\![e_1]\!](s), [\![e_2]\!](s))$, where the former upd corresponds to the operator, and the later to the function, defined earlier, used to update the contents of an array. For convenience the operator is overloaded, as happens with the arithmetic operations. This operator is only part of the assertion language and cannot be used in annotations.

Remarks:

- The inference system from Figure 2.3 is different from the standard Hoare inference system to allow the derivation of proofs in a deterministic way. Note that it does not include the consequence rule, avoiding the ambiguity created by this rule, and satisfies the sub-formula property. In fact, the definition of such system is inspired by [48] and its purpose is allowing the use of VCGens;

- Just like in [48] we also consider that loop invariants must be provided as inputs to the program verification process. For this reason the *while* rule already includes the invariant annotation;

- When a safety-sensitive Hoare triple $\{\!|\varphi|\!\}\ C\ \{\!|\psi|\!\}$ is not derivable using this system, it does not mean that the C program (without annotations) is not correct; it just means that assuming φ before executing C is not sufficient to prove that ψ holds at the end of its execution;

Soundness of the axiomatic semantics The axiomatic semantics defined by the rules and axioms of Figure 2.3 is sound relatively to the operational semantics of the *While^C* language (defined by the rules of Figure 2.2). Thus, any safety-sensitive Hoare triple that is derivable from the rules and axioms of Figure 2.3, is indeed valid [48]. Lemma 3 formally establishes this result. The proof of this lemma requires proving that the validity of $\mathsf{safe}(\mathsf{e})$, for some expression e, implies that e does not evaluate to an **error** value in any state (and vice-versa). This result is established by the following lemma.

Lemma 2. *Let e be an expression in While^C.*

$$\forall\ s \in \mathsf{State}.\ [\![\mathsf{safe}(\mathsf{e})]\!]_{\mathcal{M}}(s) = \textbf{true} \quad \text{iff} \quad [\![e]\!](s) \neq \textbf{error}$$

Proof. By induction on the structure of e. □

Lemma 3. *(Soundness of the axiomatic semantics of $While^C$)*

Let $\{\!|\varphi|\!\}\ C\ \{\!|\psi|\!\}$ be a safety-sensitive Hoare triple in $While^C$ and $\vdash_{While^C} \{\!|\varphi|\!\}\ C\ \{\!|\psi|\!\}$ denote that $\{\!|\varphi|\!\}\ C\ \{\!|\psi|\!\}$ is derivable using the axioms and rules from the inference system of Figure 2.3.

$$\vdash_{While^C} \{\!|\varphi|\!\}\ C\ \{\!|\psi|\!\} \implies \forall s \in \mathsf{State}.\ [\![\{\!|\varphi|\!\}\ C\ \{\!|\psi|\!\}]\!]_{\mathcal{M}}(s) = \mathbf{true}$$

Proof. By induction on the structure of C. We will only present the proof for $\mathsf{a}[e_1] := e_2$ and for **while** $\{\theta\}$ (e) $\{C\}$, since these are the more interesting cases.

- $\mathsf{a}[e_1] := e_2$

  By Definition 1, $[\![\{\!|\varphi|\!\}\ \mathsf{a}[e_1] := e_2\ \{\!|\psi|\!\}]\!]_{\mathcal{M}}(s) = \mathbf{true}$ if and only if $([\![\varphi]\!]_{\mathcal{M}}(s) = \mathbf{true} \wedge (\mathsf{a}[e_1] := e_2, s) \Downarrow s') \implies s' \neq \mathbf{error} \wedge [\![\psi]\!]_{\mathcal{M}}(s') = \mathbf{true}$. Let us assume that $([\![\varphi]\!]_{\mathcal{M}}(s) = \mathbf{true} \wedge (\mathsf{a}[e_1] := e_2, s) \Downarrow s')$. According to the operational semantics of $While^C$, either $s' = \mathbf{error}$, if $[\![\mathsf{a}[e_1]]\!](s) = \mathbf{error}$ or $[\![e_2]\!](s) = \mathbf{error}$, or $s' = s[\mathsf{a} \leftarrow \mathsf{upd}(s(\mathsf{a}), [\![e_1]\!](s), [\![e_2]\!](s))]$, if $[\![\mathsf{a}[e_1]]\!](s) \neq \mathbf{error}$ and $[\![e_2]\!](s) \neq \mathbf{error}$.

  Assuming that $\vdash_{While^C} \{\!|\varphi|\!\}\ \mathsf{a}[e_1] := e_2\ \{\!|\psi|\!\}$, then by the axioms and rules of the inference system of Figure 2.3, $[\![\varphi \to \mathsf{safe}(\mathsf{a}[e_1])]\!]_{\mathcal{M}}(s) = \mathbf{true}$, $[\![\varphi \to \mathsf{safe}(e_2)]\!]_{\mathcal{M}}(s) = \mathbf{true}$ and $[\![\varphi \to \psi[\mathsf{upd}(\mathsf{a}, e_1, e_2)/\mathsf{a}]]\!]_{\mathcal{M}}(s) = \mathbf{true}$. So, by Lemma 2, follows that $[\![\mathsf{a}[e_1]]\!](s) \neq \mathbf{error}$ and $[\![e_2]\!](s) \neq \mathbf{error}$, hence $s' = s[\mathsf{a} \leftarrow \mathsf{upd}(s(\mathsf{a}), [\![e_1]\!](s), [\![e_2]\!](s))]$. Since syntactic and semantic substitution are compatible, and by definition $[\![\mathsf{upd}(\mathsf{a}, e_1, e_2)]\!](s) = \mathsf{upd}(s(\mathsf{a}), [\![e_1]\!](s), [\![e_2]\!](s))$, follows that $[\![\psi]\!]_{\mathcal{M}}(s') = \mathbf{true}$.

- **while** $\{\theta\}$ (e) $\{C\}$

  Again, by Definition 1, $[\![\{\!|\varphi|\!\}\ \mathbf{while}\ \{\theta\}\ (e)\ \{C\}\ \{\!|\psi|\!\}]\!]_{\mathcal{M}}(s) = \mathbf{true}$ if and only if $([\![\varphi]\!]_{\mathcal{M}}(s) = \mathbf{true} \wedge (\mathbf{while}\ \{\theta\}\ (e)\ \{C\}, s) \Downarrow s') \implies s' \neq \mathbf{error} \wedge [\![\psi]\!]_{\mathcal{M}}(s') = \mathbf{true}$. Let us assume that $([\![\varphi]\!]_{\mathcal{M}}(s) = \mathbf{true} \wedge (\mathbf{while}\ \{\theta\}\ (e)\ \{C\}, s) \Downarrow s')$, we need to prove that $s' \neq \mathbf{error} \wedge [\![\psi]\!]_{\mathcal{M}}(s') = \mathbf{true}$. Assuming now that $\vdash_{While^C} \{\!|\varphi|\!\}\ \mathbf{while}\ \{\theta\}\ (e)\ \{C\}\ \{\!|\psi|\!\}$, then by the rules of the axiomatic semantics of $While^C$, we have that

  1. $\vdash_{While^C} \{\!|\theta \wedge e|\!\}\ C\ \{\!|\theta|\!\} \implies \forall s \in \mathsf{State}.\ [\![\{\!|\theta \wedge e|\!\}\ C\ \{\!|\theta|\!\}]\!]_{\mathcal{M}}(s) = \mathbf{true}$ (by induction hypothesis) and

2. $(\varphi \to \theta \wedge \mathsf{safe(e)}) \wedge (\theta \wedge \mathsf{safe(e)} \wedge \neg e \to \psi)$;

According to the operational semantics of *While^C*, we have three possibilities for $(\textbf{while}\ \{\theta\}\ (e)\ \{C\}, s) \Downarrow s'$: either $[\![e]\!](s) = \textbf{error}$ which implies that $s' = \textbf{error}$; or $[\![e]\!](s) = \textbf{true}$ and $(C, s) \Downarrow s'' \wedge (\textbf{while}\ \{\theta\}\ (e)\ \{C\}, s'') \Downarrow s'$; or $[\![e]\!](s) = \textbf{false}$ which implies that $s' = s$. By (2) we have that $\varphi \to \mathsf{safe(e)}$. So by Lemma 2 we conclude that $[\![e]\!](s) \neq \textbf{error}$ and consequently, $s' \neq \textbf{error}$. Now let us assume that $[\![e]\!](s) = \textbf{false}$. Since by (2), $\varphi \to \theta$ and $(\theta \wedge \mathsf{safe(e)} \wedge \neg e \to \psi)$, using Lemma 2, we immediately conclude that $[\![\psi]\!]_{\mathcal{M}}(s') = \textbf{true}$.

Finally, considering that $[\![e]\!](s) = \textbf{true}$ and $(C, s) \Downarrow s''$ and $(\textbf{while}\ \{\theta\}\ (e)\ \{C\}, s'') \Downarrow s'$ and assuming that the loop terminates in n iterations, we can use the well-known equivalence,

$$\textbf{while}\ \{\theta\}\ (e)\ \{C\} \equiv \textbf{if}\ (e)\ \{C; \textbf{while}\ \{\theta\}\ (e)\ \{C\}\}\ \textbf{else}\ \{\textbf{skip}\}$$

to unfold the loop $n+1$ times, where in the last iteration we have $[\![e]\!](s_{n+1}) = \textbf{false}$ and $s' = s_{n+1}$. For instance in the first iteration we have

$$(\textbf{while}\ \{\theta\}\ (e)\ \{C\}, s) \Downarrow s' \equiv (\textbf{if}\ (e)\ \{C; \textbf{while}\ \{\theta\}\ (e)\ \{C\}\}\ \textbf{else}\ \{\textbf{skip}\}, s) \Downarrow s'$$

which implies that for some state s_0

$$[\![e]\!](s) = \textbf{true} \wedge (C, s) \Downarrow s_0 \wedge (\textbf{while}\ \{\theta\}\ (e)\ \{C\}, s_0) \Downarrow s'$$

and using the induction hypothesis, we can conclude that $[\![\theta]\!]_{\mathcal{M}}(s_0) = \textbf{true}$. In the second iteration, we will have

$$(\textbf{while}\ \{\theta\}\ (e)\ \{C\}, s_0) \Downarrow s' \equiv (\textbf{if}\ (e)\ \{C; \textbf{while}\ \{\theta\}\ (e)\ \{C\}\}\ \textbf{else}\ \{\textbf{skip}\}, s_1) \Downarrow s'$$

for some state s_1. Using again the induction hypothesis and the fact that $[\![\theta]\!]_{\mathcal{M}}(s_0) = \textbf{true}$, we can conclude that $[\![\theta]\!]_{\mathcal{M}}(s_1) = \textbf{true}$. So we can repeat this process until iteration $n + 1$, where $[\![e]\!](s_{n+1}) = \textbf{false}$ and $[\![\theta]\!]_{\mathcal{M}}(s_{n+1}) = \textbf{true}$. But since $\theta \wedge \mathsf{safe(e)} \wedge \neg e \to \psi$, we can conclude that $[\![\psi]\!]_{\mathcal{M}}(s_{n+1}) = \textbf{true}$. Notice that by Lemma 2 we can conclude that $[\![\mathsf{safe(e)}]\!](s') = \textbf{true}$, since we have assumed that $[\![e]\!](s') = \textbf{false}$.

□

$$
\begin{array}{ll}
\mathsf{vc}(\textbf{skip}, \psi) & = \textbf{true} \\
\mathsf{vc}(\mathtt{x} := e, \psi) & = \textbf{true} \\
\mathsf{vc}(\mathsf{a}[e_1] := e_2, \psi) & = \textbf{true} \\
\mathsf{vc}(\textbf{if}\ (e)\ \{C_1\}\ \textbf{else}\ \{C_2\}, \psi) & = \mathsf{vc}(C_1, \psi) \wedge \mathsf{vc}(C_2, \psi) \\
\mathsf{vc}(\textbf{while}\ \{\theta\}\ (e)\ \{C\}, \psi) & = \mathsf{vc}(C, \theta \wedge \mathsf{safe(e)})\ \wedge (\theta \wedge \mathsf{safe(e)} \wedge \neg e \to \psi) \wedge \\
 & \quad (\theta \wedge \mathsf{safe(e)} \wedge e) \to \mathsf{wp}(C, \theta \wedge \mathsf{safe(e)}) \\
\mathsf{vc}(C_1; C_2, \psi) & = \mathsf{vc}(C_1, \mathsf{wp}(C_2, \psi)) \wedge\ \mathsf{vc}(C_2, \psi) \\
\mathsf{vc}(\textbf{assert}\ \theta, \psi) & = \textbf{true}
\end{array}
$$

$$
\begin{array}{ll}
\mathsf{wp}(\textbf{skip}, \psi) & = \psi \\
\mathsf{wp}(\mathtt{x} := e, \psi) & = \mathsf{safe(e)} \wedge \psi[e/\mathtt{x}] \\
\mathsf{wp}(\mathsf{a}[e_1] := e_2, \psi) & = \mathsf{safe}(\mathtt{a}[e_1]) \wedge\ \mathsf{safe}(\mathsf{e}_2) \wedge\ \psi[\mathsf{upd}(\mathsf{a}, e_1, e_2)/\mathsf{a}] \\
\mathsf{wp}(\textbf{if}\ (e)\ \{C_1\}\ \textbf{else}\ \{C_2\}, \psi) & = \mathsf{safe(e)} \wedge\ (e \to \mathsf{wp}(C_1, \psi)) \wedge\ (\neg\ e \to \mathsf{wp}(C_2, \psi)) \\
\mathsf{wp}(\textbf{while}\ \{\theta\}\ (e)\ \{C\}, \psi) & = \theta \wedge \mathsf{safe(e)} \\
\mathsf{wp}(C_1; C_2, \psi) & = \mathsf{wp}(C_1, \mathsf{wp}(C_2, \psi)) \\
\mathsf{wp}(\textbf{assert}\ \theta, \psi) & = \theta \wedge \psi
\end{array}
$$

Figure 2.4: Inductive definition of wp and vc for *While*^{C}

Verification condition generator The Verification Condition Generator (VCGen) for *While*^{C} is defined by:

$$\mathsf{VCGen}(\{\!|\varphi|\!\}\ C\ \{\!|\psi|\!\}) = (\varphi \to \mathsf{wp}(C, \psi)) \wedge \mathsf{vc}(C, \psi)$$

where vc corresponds to the function which retrieves the verification conditions and wp corresponds to the function which computes the weakest precondition[1] [40] for annotated *While*^{C}. Once again, these functions are defined in accordance with the operational (error) semantics we have defined for the *While*^{C} language. Their inductive definition is presented in Figure 2.4.

[1]Note that the weakest precondition of the annotated program is an approximation to the weakest precondition of the original program.

2.2 *While** language

For the purpose of this thesis, we also consider *While**, an abstraction of the Jessie language (whose features are detailed in Chapter 3). *While** includes logic types that abstract CAO/*While*^{C} types. For instance, the array data type is defined as a logic type and its axiomatisation intends to capture the properties of the CAO/*While*^{C} array operations. Such as in Jessie, we do not intend to execute *While** programs, therefore its operational semantics is not defined (we will only be concerned with its axiomatic semantics).

*While** is an imperative language that includes unbounded integers, booleans and arrays of arbitrary size, and exhibits a syntax very similar to that of *While*^{C}.

$$\begin{array}{llcl}
\text{Integers} & \mathsf{Int} \ni e^i & ::= & \mathbf{n} \mid \mathtt{x}^i \mid e^i + e^i \mid e^i - e^i \mid e^i * e^i \mid e^i \mathbin{/} e^i \mid \mathrm{get}(\mathsf{a}, e^i) \\
\text{Booleans} & \mathsf{Bool} \ni e^b & ::= & \mathbf{true} \mid \mathbf{false} \mid \mathtt{x}^b \mid e^b == e^b \mid e^b \mathrel{!=} e^b \mid e^i < e^i \mid e^i > e^i \\
\text{Commands} & \mathsf{Comm} \ni C & ::= & \mathbf{skip} \mid \mathtt{x} := e \mid \mathsf{a} := \mathrm{set}(\mathsf{a}, e^i_1, e^i_2) \mid \\
 & & & \mathbf{if}\ (e^b)\ \{C_1\}\ \mathbf{else}\ \{C_2\} \mid \mathbf{while}\ (e^b)\ C \mid C_1\ ;\ \ C_2
\end{array}$$

Integer literals are ranged by **n** and boolean literals are **true** and **false**. Integer and boolean variables are ranged by $\mathtt{x}^i$ and $\mathtt{x}^b$, respectively, and array variables are ranged by a. Again, whenever it is implicit from the context, we will omit the superscripts i and b when referring to integer and boolean expressions, and e will be used to denote expressions in general. Integer variables are interpreted as (unbound) integers ($\mathbb{Z}$) and booleans as values of {**true**, **false**}. Array variables are mapped to values of a type A with the following operations:

$$\mathrm{get} : A \times \mathbb{Z} \to \mathbb{Z}$$
$$\mathrm{set} : A \times \mathbb{Z} \times \mathbb{Z} \to A$$

where get(a, i) returns the element stored in the position with subscript i of array a and set(a, i, e) stores the value e in the position with subscript i of array a. These operations are constrained by the following axioms:

$$\forall\, a\ e_1\ e_2.\ \mathrm{get}(\mathrm{set}(\mathsf{a}, e_1, e_2), e_1) = e_2$$
$$\forall\, a\ e_1\ e_2\ e_3.\ e_1 \neq e_3 \to \mathrm{get}(\mathrm{set}(\mathsf{a}, e_1, e_2), e_3) = \mathrm{get}(\mathsf{a}, e_3)$$

$$\text{(skip)}\frac{}{\{\varphi\}\ \textbf{skip}\ \{\psi\}}\ \mathsf{if}\ \varphi \rightarrow \psi \qquad \text{(assign)}\frac{}{\{\varphi\}\ \mathtt{x} := e\ \{\psi\}}\ \mathsf{if}\ \varphi \rightarrow \psi[e/\mathtt{x}]$$

$$\text{(assign-array)}\frac{}{\{\varphi\}\ \mathsf{a} := \mathsf{set}(\mathsf{a}, e_1, e_2)\ \{\psi\}}\ \mathsf{if}\ \varphi \rightarrow \psi[\mathsf{set}(\mathsf{a}, e_1, e_2)/\mathsf{a}]$$

$$\text{(if-else)}\frac{\{e \wedge \varphi\}\ C_1\ \{\psi\} \qquad \{\neg e \wedge \varphi\}\ C_2\ \{\psi\}}{\{\varphi\}\ \textbf{if}\ (e)\ \{C_1\}\ \textbf{else}\ \{C_2\}\ \{\psi\}} \qquad \text{(seq)}\frac{\{\varphi\}\ C_1\ \{\theta\} \qquad \{\theta\}\ C_2\ \{\psi\}}{\{\varphi\}\ C_1;\ C_2\ \{\psi\}}$$

$$\text{(while)}\frac{\{\theta \wedge e\}\ C\ \{\theta\}}{\{\varphi\}\ \textbf{while}\ \{\theta\}\ (e)\ \{C\}\ \{\psi\}}\ \mathsf{if}\ (\varphi \rightarrow \theta) \wedge (\theta \wedge \neg e \rightarrow \psi)$$

$$\text{(assert)}\frac{}{\{\varphi\}\ \textbf{assert}\ \theta\ \{\psi\}}\ \mathsf{if}\ \varphi \rightarrow (\theta \wedge \psi)$$

Figure 2.5: Axiomatic semantics for *While**

Hoare logic for *While**

The *Hoare logic* formal system to reason about the correctness of *While** programs is expressed in terms of *partial correctness Hoare triples* of the form $\{\varphi\}\ C\ \{\psi\}$, where φ and ψ are first-order logic predicates and C is a *While** program. Again, the predicates (also called assertions) φ and ψ are called pre- and postconditions, respectively. *While** programs are also annotated with loop invariants and general assertions.

The syntax of the assertion language is the same to that of *While*^{C} (defined in the previous section). The definition of validity of an assertion φ is given by the regular notion of validity considered in first-order logic, and is denoted by $\models_{\mathcal{T}} \varphi$, where $\mathcal{T}$ corresponds to the theory $\mathcal{T}_{\mathbb{Z}} \bigcup \mathcal{T}_{\mathsf{bool}}$ plus the set of axioms (defined above) which constrain the array operations.

Axiomatic semantics The axiomatic semantics for the *While** language is given by the rules and axioms depicted in Figure 2.5 and can be used to establish the validity of partial correctness Hoare triples.

Definition 4. *(Derivability of a partial correctness Hoare triple)*

Let $\{\varphi\}\ C\ \{\psi\}$ be a partial correctness Hoare triple in While*. We say that $\{\varphi\}\ C\ \{\psi\}$ is derivable assuming the theory $\mathcal{T}$ (defined above), denoted $\vdash_{\mathcal{T}} \{\varphi\}\ C\ \{\psi\}$, if there exists a proof of $\{\varphi\}\ C\ \{\psi\}$, using the axioms and rules of the axiomatic semantics of While* (defined in Figure 2.5), which uses as assumptions only assertions from $\mathcal{T}$.

$$
\begin{aligned}
\mathsf{vc}(\mathbf{skip}, \psi) &= \mathbf{true} \\
\mathsf{vc}(\mathtt{x} := e, \psi) &= \mathbf{true} \\
\mathsf{vc}(\mathsf{a} := \mathrm{set}(\mathsf{a}, e_1, e_2), \psi) &= \mathbf{true} \\
\mathsf{vc}(\mathbf{if}\ (e)\ \{C_1\}\ \mathbf{else}\ \{C_2\}, \psi) &= \mathsf{vc}(C_1, \psi) \wedge \mathsf{vc}(C_2, \psi) \\
\mathsf{vc}(\mathbf{assert}\ \phi, \psi) &= \mathbf{true} \\
\mathsf{vc}(\mathbf{while}\ \{\theta\}\ (e)\ \{C\}, \psi) &= \mathsf{vc}(C, \theta)\ \wedge (\neg\, e \wedge \theta \rightarrow \psi)\ \wedge (e \wedge \theta \rightarrow \mathsf{wp}(C, \theta)) \\
\mathsf{vc}(C_1; C_2, \psi) &= \mathsf{vc}(C_1, \mathsf{wp}(C_2, \psi)) \wedge \mathsf{vc}(C_2, \psi)
\end{aligned}
$$

$$
\begin{array}{ll}
\mathsf{wp}(\mathbf{skip}, \psi) = \psi & \mathsf{wp}(\mathtt{x} := e, \psi) = \psi[e/\mathtt{x}] \\
\mathsf{wp}(\mathsf{a} := \mathrm{set}(\mathsf{a}, e_1, e_2), \psi) = \psi[\mathrm{set}(\mathsf{a}, e_1, e_2)/\mathsf{a}] & \mathsf{wp}(C_1; C_2, \psi) = \mathsf{wp}(C_1, \mathsf{wp}(C_2, \psi)) \\
\mathsf{wp}(\mathbf{while}\ \{\theta\}\ (e)\ \{C\}, \psi) = \theta & \mathsf{wp}(\mathbf{assert}\ \theta, \psi) = \theta \wedge \psi \\
\mathsf{wp}(\mathbf{if}\ (e)\ \{C_1\}\ \mathbf{else}\ \{C_2\}, \psi) = (e \rightarrow \mathsf{wp}(C_1, \psi))\ \wedge & (\neg\, e \rightarrow \mathsf{wp}(C_2, \psi))
\end{array}
$$

Figure 2.6: Inductive definition of the functions vc and wpc

Verification condition generator The Verification Condition Generator (VCGen) for *While** is defined as follows:

$$\mathsf{VCG}_{\mathsf{while}^*}(\{\varphi\}\ C\ \{\psi\}) = (\varphi \rightarrow \mathsf{wp}(C, \psi)) \wedge \mathsf{vc}(C, \psi)$$

where again vc corresponds to the function which retrieves the verification conditions and wp corresponds to the function which computes the weakest precondition for annotated *While**. Their inductive definition is presented in Figure 2.6.

A *While** annotated program is correct if all of the generated verification conditions are valid. The soundness of this VCGen with respect to the axiomatics semantics can be establish by the following proposition.

Proposition 5. *(VCGen soundness)*

Let $\{\varphi\}\ C\ \{\psi\}$ *be a Hoare triple in While*.*

$$\models_{\mathcal{T}} \mathsf{VCG}_{\mathsf{while}^*}(\{\varphi\}\ C\ \{\psi\}) \quad \text{iff} \quad \vdash_{\mathcal{T}} \{\varphi\}\ C\ \{\psi\}$$

The result above implies that if all the verification conditions are valid, there always exist a derivation tree using the rules of the axiomatic semantics of *While**, whose side-conditions are valid too (and vice-versa) [48].

2.3 Security properties

Deductive verification based techniques can be used to enforce security policies in software systems implementations. This section introduces some of the security policies addressed in this thesis, together with their formalisation over single program executions.

2.3.1 Safety properties

Safety is related with preventing runtime errors which are due to not accounted situations in the evaluation semantics of the programming language. To reason about program safety, one has to consider an operational semantics which deals with such errors. The $While^C$ language, for example, captures in its operational semantics an error state. A program C written in $While^C$ is considered to be safe if for every state s, if C evaluates to a state s' then s' is different from the error state.

Definition 6. *(Safe program)*
A program C is safe, if $\forall s \in \mathsf{State}.\ ((C, s) \Downarrow s') \implies s' \neq \mathbf{error}$.

This property can be captured by *safety-sensitive* Hoare triples. The inference system for these safety-sensitive Hoare triples (already defined in Figure 2.3) does not depend only on the structure of the command, but also on expressions that may occur in it. Note that 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.

Termination Some authors consider program termination as a safety property (e.g. Frama-c). In this thesis, termination is treated separately.

Program termination is characteristic of total correctness specifications and in languages such as $While^C$ (without recursive, or such as CAO, where recursion is limited), is established by proving the termination of the loops. In this setting, the while rule includes a *loop variant* which is a program expression whose value decreases with each iteration. In this total correctness rule, presented below, V corresponds to the loop variant (also annotated in the code) and v_0 to its initial value.

$$\text{(while)}\frac{[\theta \land b \land V = v_0]\ C\ [\theta \land V < v_0]}{[\theta]\ \textbf{while}\ \{\theta, V\}\ (b)\ \{C\}\ [\theta \land \neg b]} \quad \text{if } \theta \land b \to (V \geq 0)$$

Typically, the rule expresses that the value V must decrease at each iteration and be greater or equal to zero at the very end. As such, to prove program termination one has to annotate each loop with a proper loop variant.

2.3.2 Information flow

Information flow policies are security policies concerned with controlling the way information is propagated in the system during execution. A formalisation of these properties, proposed by Denning and Denning [38], includes a set of *security classes*, a *flow relation* and a method of *binding*. Information is associated to a security class using the binding method and the flow relation is responsible for dictating how information of different classes can flow during the program execution. A program is considered to be secure if there exist no violations of the flow policy. This approach has several advantages since it can be used to formalise noninterference, an information flow policy, introduced by Goguen and Meseguer [50], which inspires the formal definition of the security policies addressed in this thesis.

Noninterference Informally, noninterference expresses that, during the program execution, private (high) inputs must not influence the computation of public (low) outputs. A possible formalisation of this intuition is based on programming-language semantics.

Consider the evaluation semantics of the While^C language and let V_H and V_L denote the sets of high-security and low-security variables of program C, respectively, where $V_L = \mathit{Vars}(C) \setminus V_H$. Let also $s_1 \overset{X}{=} s_2$ denote that the states s_1 and s_2 are X-indistinguishable, i.e., $\forall x \in X.\ s_1(x) = s_2(x)$, for some set of variables X. We consider *termination insensitive* and *termination sensitive* definitions of noninterference.

Definition 7. *(Termination-insensitive non-interference)*

A program C satisfies termination-insensitive non-interference if,

$$\forall s_1, s_2 \in \mathsf{State}.\ (s_1 \overset{V_L}{=} s_2\ \wedge\ (C, s_1) \Downarrow s_1'\ \wedge\ (C, s_2) \Downarrow s_2')\ \Longrightarrow\ s_1' \overset{V_L}{=} s_2'$$

Definition 8. *(Termination-sensitive non-interference)*

A program C satisfies termination-sensitive non-interference if,

$$\forall s_1, s_2 \in \mathsf{State}.\ (s_1 \overset{V_L}{=} s_2\ \wedge\ (C, s_1) \Downarrow s_1')\ \Longrightarrow\ ((C, s_2) \Downarrow s_2'\ \wedge\ s_1' \overset{V_L}{=} s_2')$$

Self-composition Barthe et al. [17] propose a simple methodology based on programming logics to prove that a program satisfies non-interference. The authors observe that noninterference of a program C can be reduced to a property about a single program execution of the program $C; C^r$, where C^r is a renamed copy of C. This characterization of noninterference relies on the idea of *self-composition*.

More rigorously, given some program C, let C^r be the program that is equal to C except that every variable x is renamed to a fresh variable x^r. Non-interference can be formulated considering a single execution of the self-composed program $C; C^r$. Note that any state s of $C; C^r$ can be partitioned into two states s^o and s^r with disjoint domains, such that $s = s^o \cup s^r$ and $\mathsf{dom}(\mathsf{s}^o) = \mathit{Vars}(\mathsf{C})$ and $\mathsf{dom}(\mathsf{s}^r) = \{\mathsf{x}^r | \mathsf{x} \in \mathit{Vars}(\mathsf{C})\}$ (where $\mathsf{dom}(\mathsf{s})$ retrieves the domain of the state s).

C satisfies termination-insensitive noninterference if any terminating execution of the self-composed program $C; C^r$, starting from a state s such that s^o and s^r differ only in the values of high-security variables, results in a final state s' such that s'^o and s'^r are equivalent with respect to the values of low-security variables.

Definition 9. *(Self-composition: termination-insensitive non-interference)*
Let C be a program, C^r a renamed copy of C and s a state. C satisfies termination-insensitive noninterference if,

$$\forall x \in V_L.\ (s(x) = s(x^r) \wedge (C; C^r, s) \Downarrow s') \implies s'(x) = s'(x^r)$$

Using the definition above (based on self-composition), termination-insensitive non-interference can now be formalised as a Hoare logic partial correctness specification of the form:

$$\left\{ \bigwedge_{x \in V_L} x = x^r \right\} C; C^r \left\{ \bigwedge_{x \in V_L} x = x^r \right\}$$

We remark that termination-sensitive non-interference can be captured by total correctness Hoare triples.

2.3.3 Functional correctness properties

To handle program equivalence we propose a generalisation of the *self-composition* technique [17] introduced above, called *equivalence by composition*.

Considering two programs C_1 and C_2 written in While^C, we are interested in proving their equivalence. Let V be the set of variables occurring in both programs (we admit both use the same set of variables, otherwise we may let $V = \mathit{Vars}(C_1) \cap \mathit{Vars}(C_2)$). The idea that we want to capture is that if the programs are executed from indistinguishable states with respect to V, they terminate in states that are also indistinguishable.

Definition 10. *(Program equivalence)*

Let C_1 and C_2 be programs and V a set of variables occurring in both programs. We say that C_1 is equivalent to C_2 if,

$$\forall s_1, s_2 \in \mathsf{State}.\ (s_1 \overset{V}{=} s_2\ \wedge\ (C_1, s_1) \Downarrow s_1' \wedge s_1' \neq \mathbf{error}) \implies$$
$$((C_2, s_2) \Downarrow s_2' \wedge s_2' \neq \mathbf{error} \wedge\ s_1' \overset{V}{=} s_2')$$

Observe now that the execution of C_1 and C_2 can be reduced to a property about a single program execution of the program $C_1; {C_2}^r$, where again ${C_2}^r$ is a renamed copy of C_2. Therefore C_1 and C_2 will be defined as equivalent if every execution of the composed program $C_1; {C_2}^r$, starting from a state in which the values of corresponding variables are equal, terminates in a state with the same property. Notice that any state s of $C_1; {C_2}^r$ can be seen as a disjoint union of s_1 and s_2 such that $s = s_1 \cup s_2$ and $\mathsf{dom}(\mathsf{s}_1) = \mathit{Vars}(\mathsf{C}_1)$ and $\mathsf{dom}(\mathsf{s}_2) = \mathit{Vars}({\mathsf{C}_2}^{\mathsf{r}})$.

Definition 11. *(Equivalence by composition)*

Let C_1 and C_2 be programs and V a set of variables occurring in both programs. Considering that ${C_2}^r$ is the program C_2 where all variables x are renamed to fresh x^r, we say that C_1 is equivalent to C_2 if,

$$\forall s \in \mathsf{State}.\ \forall x \in V.\ (s(x) = s(x^r)\ \wedge\ (C_1; {C_2}^r, s) \Downarrow s' \wedge s' \neq \mathbf{error}) \implies s'(x) = s'(x^r)$$

This property can be expressed as the following Hoare logic total correctness specification:

$$\left[\bigwedge_{x \in V} x = x^r\right] C_1; {C_2}^r \left[\bigwedge_{x \in V} x = x^r\right]$$

Note that the definition of program equivalence (Definition 10) requires that if C_1 terminates in a state different from the error state, then C_2 must also terminate in a non-error state, therefore this property can only be expressed using total correctness Hoare triples (as happens with termination sensitive noninterference).

Weaker notions of equivalence can be handled by taking V to be a subset of $\mathit{Vars}(C_1) \cap \mathit{Vars}(C_2)$. In fact, we are not restricted to equivalence relations – arbitrary relations can be considered between the two partitions of the state:

$$[R_1(s, s^r)]\ C_1; {C_2}^r\ [R_2(s, s^r)]$$

where s and s^r denote the state partitions associated with C_1 and ${C_2}^r$ respectively.

Notice that the specification presented above covers the original definition of self-composition [17]. Considering ${C_2}^r$ a renamed copy of C_1, and defining the following equality of predicates:

$$R_1(s, s^r)\ \equiv R_2(s, s^r)\ \equiv\ s \overset{V_L}{=} s^r$$

it is easy to see, that the definition of self-composition fits into the definition above.

Chapter 3

Related work

In the previous chapter we introduced the key concepts underlying the work of this thesis. In this chapter we present related work that contextualises these concepts. It is divided in different parts, according to the organisation of this thesis. First we explain how deductive verification can be used in the context of language-based security. We then introduce deductive verification related concepts and the deductive verification platform included in the Frama-c framework. We conclude by presenting some work that has been done to enforce security properties akin to the ones studied in this thesis, namely information flow policies, functional correctness and non-functional properties, using language-based approaches.

3.1 Language-based security

Language based security is the area that addresses the study of formal techniques to enforce security in software at programming language level [59, 83]. The work carried out in this area, follows two main approaches: *Dynamic* techniques enforce the security requirements during the program execution (runtime); *Static* techniques are based on the verification of security requirements at compile-time.

Dynamic techniques usually introduce an overhead on program execution, although in some cases they may be the only feasible way to enforce conformance to a security policy. *In-lined reference monitors* (IRM) are one of the techniques that rely on dynamic analysis. This technique resorts to program rewriting to embed a reference monitor in the target system. The *reference monitor* dynamically observes program execution and intervenes whenever it is close to violating some security policy. The behavior of an IRM can be defined through a *security automata* residing somewhere between the program and the machine where the program executes. The automata examines a sequence of actions that are security relevant and terminates program execution when it detects an action that violates some security policy. Some work on this direction is presented in [22, 82, 51, 61].

On the other hand, static analysis techniques do not modify the performance of the target program. These techniques can be themselves split according to the language level at which they operate: the *semantic* level or *type system* level [59, 83]. Language-based techniques applied at the type system level, are based on writing the program in conformance with a type system. Type-checking can be applied at different levels of the compilation process: at high-level or at low-level. At high-level, the security is enforced by type-checking the source program written in a high-level language [72, 74, 73, 93, 94]. On the other hand, at low-level, security is granted by type-checking an augmented assembly language/object-code with a type system [98, 20, 19, 69]. Usually, type systems used in this approach assume a particular form of a collection of typing rules which describe what security type is assigned at each program expression. A disadvantage on the use of custom type systems is that for each new variation of the security policy and programming language, the type system needs to be redefined and proven sound.

Techniques applied at the semantic level model security as a program behavior property. Deductive verification techniques, the ones we address in this thesis, fit into this classification: security policies are expressed using contract-based specification and deductive verification platforms are used to verify if the software systems satisfy them [57, 95, 17, 88, 76, 42, 23]. Several reasons led us to adopt deductive verification techniques in this work: unlike dynamic techniques, they do not introduce an overhead on program execution; a single formalism can be used to verify and specify a relevant set of security properties; the existence of huge set of deductive verification tools for general purpose languages, such as C or Java. The next section details the most important concepts about deductive verification. We emphasize that there are also other techniques based on semantic models which rely on other formal models such as temporal logic [36], for example.

3.2 Deductive verification

Formal verification (or program verification) aims to study mathematically based methodologies to validate if a program rigorously satisfies a given property or a pre-defined formal specification of the system. Different mathematical models are usually used to abstract the systems behaviour, depending on the systems nature, giving rise to different approaches, such as *deductive verification* [54, 47], *abstract interpretation* [35], *model checking* [43] and *symbolic execution* [29]. In this work, we focus on *deductive verification* based approaches.

Hoare logic Deductive verification is a formal approach which relies on *Hoare logic* [54, 47] to model the systems behaviour. *Hoare logic* is formal system introduced by C.A.R. Hoare [54] in 1968, with a specific notation – the *Hoare triples* – to specify the desired behaviour of the programs. A *Hoare triple* for a given command C is a specification of the form $\{\varphi\}\ C\ \{\psi\}$, where:

- φ, called *precondition*, corresponds to a predicate, in first-order logic, on the initial values (initial state) of program variables. The validity of this condition implies the correct execution of the program. If the precondition is not verified before the program executes, it can behave in unexpected ways;
- ψ, called *postcondition*, corresponds to a predicate (also in first-order logic) on the final values (final state) of program variables. If the predicate is valid, then it implies that the program satisfies the specification.

The meaning of the logical predicate $\{\varphi\}\ C\ \{\psi\}$ is as follows: if φ holds before C execute, and if C terminates, then ψ must be valid after the execution. This standard specification is *partial*, because for the predicate $\{\varphi\}\ C\ \{\psi\}$ to be true, it is not required that C terminates, when started in state where φ is valid. The fundamental requisite is that if C terminates, then ψ must be valid. Nevertheless, there is a stronger kind of specification, usually represented by $[\varphi]\ C\ [\psi]$ and called *total correctness specification*, which is true if and only if two conditions are valid:

- whenever C is executed in a state satisfying φ, then it terminates;
- after termination, ψ holds in the final state.

The usual method to validate if the program is according with the specification, is to use a sound *inference system.* This defines the axiomatic semantics of the underlying language, which is usually seen as an alternative to the semantics of the programming language. A proof in Hoare logic is then a sequence of lines, each of which is either an axiom or follows from earlier lines by an inference rule. These are usually presented as trees, called *derivation trees* (or *proof trees*). Therefore, verifying if a Hoare triple $\{\varphi\}\ C\ \{\psi\}$ is valid in this system, implies constructing a derivation tree whose conclusion is $\{\varphi\}\ C\ \{\psi\}$ and all the side-conditions evaluate to true [8].

Verification condition generators Proofs using inference systems are typically long and tedious to write. For this reason, many attempts have been made to mechanise the proof, by designing systems to partially handle formal proof generation in Hoare logic. Even if those systems cannot prove everything automatically, it is possible to check whether an arbitrary formal proof is valid.

When the inference system is well behaved (there is no ambiguity in the choice of the rule to apply) and satisfies the *sub-formula* property (all assertions that occur in premises also occur in conclusions) one can define a strategy to derive proofs in a deterministic way. The usual approach taken in these cases is the use of *verification conditions generators* (VCGens), which are systems that generate a set of purely mathematical statements called *proof obligations* (also known as *verification conditions*) from annotated specifications (with pre- and postconditions). The validity of the verification conditions can be established by an external proof tool, such as theorem prover (e.g.: Simplify, Alt-Ergo, Z3, etc) or a proof assistant (e.g.: Coq, Isabelle, PVS, etc).

Summing up, in the past few years a lot of research has been done to automate the generation of proof obligations for general-purpose languages, and verification platforms based on variations of Hoare logic came up. These platforms tend to speed up the proof process, by providing means to automate the generation of annotations and associated verification conditions. They are usually structured around the following components:

- **Annotation Language**: the specification (Hoare triples specifications) is included in the source code by means of program annotations. Besides pre- and postconditions, annotated programs include loop annotations and general assertions, intended to facilitate the proof process. Annotations are embedded in the code [62] or inserted as comments with a special notation [21, 14];
- **Verification condition generator (VCGen)**: this is responsible for taking the annotated program and generate the proof obligations;
- **Proof generation**: proof obligations are essentially formulas in first-order logic, so that a first-order proof tool (an automatic prover such as Simplify, or a proof

assistant such as Coq) is required to construct the proof that the formulas are valid. Their validity will imply that the software is indeed correct with respect to the specification.

The next section details the most important aspects of the deductive verification platform included in the Frama-c framework. We also highlight other deductive verification platforms, noting that their architectures are very close to that of Frama-c.

3.3 Frama-C

Frama-c is a framework for the static analysis of C programs which includes an off-the-self deductive verification tool. It allows static analyzers implemented as plug-ins, enabling a fine-grained collaboration of analysis techniques between them. For example, a new plug-in may rely on the functionalities and results of existing plug-ins. At the moment, there are three important plug-ins that almost all of the other plug-ins rely on: the value analysis plug-in, the Jessie plug-in and the weakest precondition plug-in. The first one is based on abstract interpretation and computes supersets of possible values for expressions on the program. The Jessie plug-in and the weakest precondition plug-in can be used for deductive verification of C programs. In our work we are focused on the Jessie plug-in. This plug-in can be seen as a translator between Frama-c and the Why platform, as depicted in Figure 3.1.

3.3.1 ACSL

ACSL [21] is the behavioral interface specification language of Frama-c and is mostly inspired by the specification language of Caduceus [44] which is itself inspired by JML [62]. The main difference between ACSL and JML lies on the fact that ACSL is only focused in the static analysis of C programs, whereas JML aims both at runtime assertion checking and static analysis of Java programs [46].

ACSL annotations are introduced in the C source files as comments with a special notation (start with `/*@` or `//@`). Annotations are classified as *global* or *statement* annotations. Function contracts (pre- and postconditions) and global invariants are considered to be global annotations. On the other hand, assertions and loop annotations are said to be statement annotations.

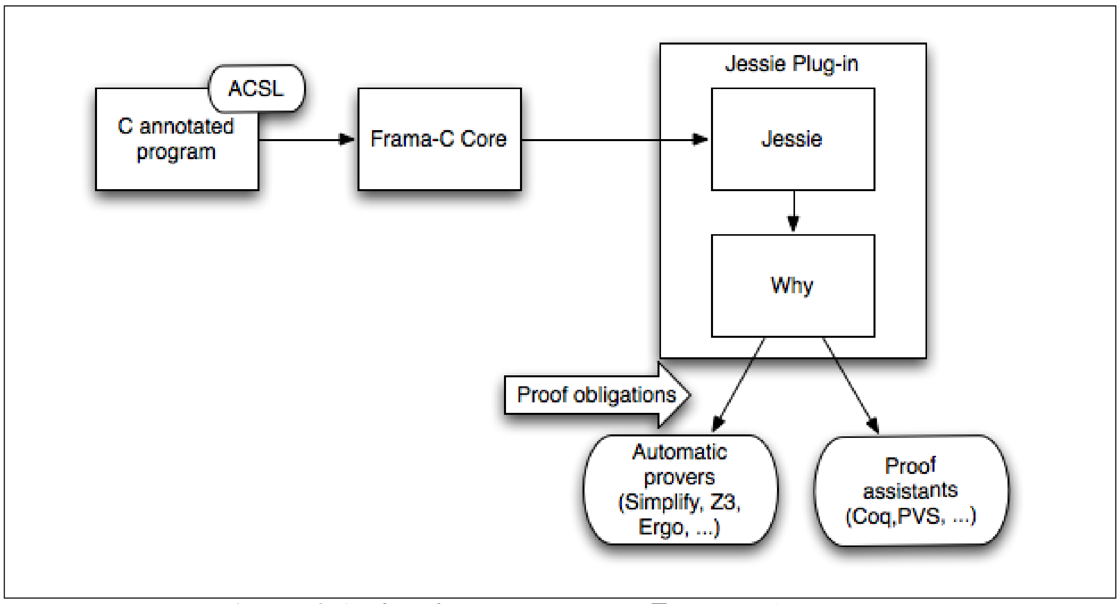

Figure 3.1: Jessie plug-in of the Frama-c framework

Besides standard contract specifications, the programmer can also annotate the program with *logic specifications* (global annotation) and *ghost code* (statement annotation). Logic specifications are definitions of logic functions or predicates, lemmas and axiomatisations. The latter is very useful to include the axiomatisation of new logic types, logic functions and predicates by defining the axioms they satisfy. *Ghost code* is regular C code only allowed to modify ghost variables and only visible in annotations. Appendix B contains examples of C source files annotated with ACSL specifications.

3.3.2 Jessie plug-in

The verification process using the Jessie tool consists of analysing the annotated C program and generating a set of proof obligations using the Why VCGen [45] (introduced in the following section). The first step of the verification process is to conveniently annotate the C programs using the behavioural interface specification language ACSL. After verifying that the C annotated code is syntactically correct, the Frama-c Core translates it to the Jessie input language. The Jessie plug-in is then responsible for translating the output of the Frama-c Core into Why. Afterwards, Why is used to generate the proof obligations than can be proved by a multitude of proof tools. The validity of these proof obligations implies that the source program satisfies the specified set of properties. In addition to the specification annotated into the program, the Jessie plug-in automatically generates proof obligations that imply the safeness of the underlying code.

Safety here is related with memory safety, arithmetic safety and program termination.

3.3.3 Why platform

Why [45] is a software verification platform developed by the ProVal team and is currently in its third version: *Why3*. The Why tool is based on the *Weakest Precondition* calculus [40] to derive the proof obligations. Nevertheless, it is not a proof tool: it produces proof obligations for existing external proof tools such as automatic provers (like Simplify, Z3, etc) or proof assistants (like Coq, PVS or Isabelle). Furthermore, Why is easily extensible to target other automatic prover or proof assistant back-end. The Why input language consists of a programming language very similar to ML, with imperative features (references and arrays), exceptions and annotations. Typing rules totally exclude aliasing between different mutable variables. Why supports exceptions that can be declared by the user, and the notion of `effects` (e.g. `read` or `write`), which indicate a possible raise of an exception.

Although Why can be directly used to verify programs, the general purpose of Why is to be used as a back-end by other verification tools. As we will see further in this chapter, besides the Jessie plug-in, there are also other verification tools which rely on the Why tool.

3.3.4 Jessie input language

Jessie is only used in the verification of C programs in the Frama-c framework and programmers are not expected to produce Jessie source programs from the scratch. However, Jessie itself can be seen as a separate tool with its own input language. The Jessie input language is a simply typed imperative language, developed in parallel with ACSL, from which it inherits many of its constructions.

The language combines operational and logical features, very much like *While** introduced in the previous chapter. The operational part refers to statements which describe the control flow and instructions that perform operations on data, including memory updates. The logical part consists of first-order logic formulas, attached to statements and functions in the form of annotations. Jessie provides primitive types, abstract datatypes, and also allows the definition of new datatypes. Programs can be annotated using pre- and postconditions, loop invariants, and other intermediate assertions. The logical language is typed and includes built-in structural equality.

Syntax The abstract syntax defined in [70] includes the syntax of types, terms, statements and global variables and functions. In Jessie a type is either a base type or a user defined type. Base types are mathematical integers, booleans and real numbers plus unit (the void type). User-defined types can be either limited range integers, structured types such as arrays and structs, or pointers to structured types. Jessie terms include variables, memory locations (pointer access or access to fields of structs), constants and integer and real literals. Operations such like integer/real arithmetic, integer/real comparison, pointer operations (pointer arithmetic, difference and comparison), boolean operations, unary and binary operations and cast operations are also taken into account. Jessie statements are structured around conditionals and loops. It also includes a mechanism to manage exceptions, which allows for exceptional control flow. A special `main` function, with `unit` return type and no parameters is considered to be the entry point of the program. Unlike in C, in Jessie there is no implicit initialization of variables. In this thesis we only take advantage of some of these constructions (such as some of the Jessie base types – integers, booleans and unit – and some of the Jessie statements).

Annotation language Additional constructs are included in Jessie in the form of an *annotation language* that allows reasoning about the execution of Jessie programs. Since Jessie programs are not meant to be executed, annotations appear naturally in the program, contrary to happens with ACSL annotations. The inheritance from ACSL is more clear in the annotation language. It includes the definition of logic types (user-defined types that can be used in the program); logical terms, including logical function application and range location; first-order logic propositions (comparison, conjunction, disjunction and implication); definition of logical functions, predicates and lemmas. Moreover, just like in ACSL, one can also define new logic types, logic functions and predicates by means of axioms. This is perhaps the most interesting feature of the Jessie language, since it allows the introduction of new types and first-order logic theories associated to them, as we will see in Chapter 6.

3.3.5 Other deductive verification platforms

The verification infrastructure introduced in the Jessie plug-in and also in the CAOVerif tool (the one we propose in this thesis in Chapter 6), was already used in the development of other verification tools for C and Java. Caduceus [44] (currently deprecated), a tool for C, and Krakatoa [67], a tool for Java, are also built on the top of Why tool. The main difference in the infrastructure of Caduceus and Krakatoa is that the first one compiles directly into Why and Krakatoa compiles to Jessie first (and then uses the Jessie plug-in to generate the verification conditions). Caduceus [44] is the predecessor of the Jessie plug-in. Just like in Jessie, in addition to the specification annotated into the program, the tool automatically generates proof obligations that imply the safety of the program with respect to absence of null pointer dereferencing and out-of-bounds array accesses.

Boogie [13] is a verification condition generator very similar to Why. The input languages to Boogie and Why are both languages with imperative features (such as goto statements in Boogie and conditional branches in Why) and first-order propositions. In both cases, the generation of verification conditions is based on the weakest precondition calculus [40]. Boogie has front-ends for extensions to C# and C which enrich the languages with annotations in first-order logic, such as pre- and postconditions, assertions and loop invariants. The C# extension, known as Spec# [14], corresponds to a superset of the C# language in the .Net framework. The Spec# tool-chain encompasses different transformations of the source code in order to get the verification conditions. Firstly, the Spec# compiler, from an annotated C# program, generates the bytecode written in the Common Intermediate Language of the .Net framework (CIL). This bytecode is then translated to BoogiePL, the input language for Boogie. At the very end, Boogie performs loop-invariant inference using abstract interpretation and finally generates the verification conditions for Simplify [39] or Z3 [37].

VCC [30, 25] is a deductive verification tool for low-level concurrent C programs where the core component is also Boogie. VCC includes a C front-end which from an annotated program, generates BoogiePL. Boogie then generates the verification conditions for an automatic prover or to HOL-Boogie, a verification environment based on VCC/Boogie and Isabelle/ HOL.

Esc/Java [46] is another deductive verification tool for Java programs whose annotation language is a subset of JML [62]. Its architecture is similar to the ones presented above and based on the earlier checker for the Modula-3 language. The tool is composed by a front-end, that parses and type-checks the annotated Java program, a translator and a VCGen. The translator is responsible for producing the VCGen input, from the parsed and type-checked program. The VCGen is also based on Dijkstra's weakest precondition calculus [40] and produces the verification conditions to the theorem prover Simplify [39]. The particularity of this tool is that it performs loop unrolling and the verification conditions generation include optimizations to avoid the exponential blow-up inherent in a naive weakest-precondition computation. It looks for run-time errors in annotated Java programs, but it does not model arithmetic overflow.

Jack (Java Applet Correctness Kit) [16] is a static verification tool for JML-annotated programs. It provides support for annotation generation and for interactive verification of functional specifications, as well as support for verification of byte-code programs. Its integration in EclipseIDE[1] allows proof obligation inspection and visualising where in code they are originated. Such like Caduceus and Frama-c, Jack is prover independent and supports an automatic (Simplify) and interactive prover (Coq).

3.4 Information flow

As we will see in the following chapters (Chapter 4 and Chapter 5), some of the security properties addressed in this thesis are classified as information flow policies. Information flow policies are security policies which aim to control the way information is propagated in the system during its execution. Extensive work has been done in the study of language-based techniques to enforce these properties in software systems. This is due to the fact that important security requirements, such as confidentiality and integrity, can be modeled as information flow policies. In fact, for confidentiality and integrity requirements, two different models, but at the same time dual, are widely used. Bell et. all proposed the well-known Bell-LaPadula[60] model, which is related with enforcement of confidentiality and Biba[24] proposed a model to enforce integrity. In both models, information is classified through security levels, from *public* to *top secret*, and entities that have access to information are also assigned to security levels. These security levels are partial ordered forming a lattice, known as the *security lattice*. This lattice corresponds to the flow relation introduced by Denning and Denning in [38].

Information flow type-systems Attempts to model information flow policies using information flow type-systems were first proposed by Volpano et al. [93]. These correspond to augmented type-systems where every program expression has a security type with two parts: an ordinary type, e.g.: int, and a label that expresses an information flow policy on the uses of labeled data. The label corresponds to the security level that is associated to the data and typing rules establish the flow relation. For example, if

[1]www.eclipse.org

the variables that store public (low) information are classified as low (L), and variables that store secret (high) information are classified as high (H), a flow relation to enforce confidentiality of secret information will express that information is only allowed to flow from L to H. A survey on this subject can be found at [80]. Information flow type-systems for low-level languages were also proposed [98, 20, 19], nevertheless low-level languages exhibit some problems that must be overcome in order to use this technique in realistic scenarios [98].

The main challenge in designing information flow type-systems is that they are often too conservative in practice, so that secure programs may be rejected. To illustrate this point, consider a security lattice with only two levels: public/low (L) and private/high (H) and the confidentiality policy defined above. Consider now the following code:

```
l := h;
l := 0;
```

where l is a variable classified as low (L) and h a variable classified as high (H).

Recall the definition of non-interference introduced in the previous chapter. Typically, confidentiality can be formalised as a non-interference property such that it states that executing the program twice, for the same low input values, the program must produce the same low output values. Under this definition the program above is indeed secure. However, type-checking the program using an information flow type-system will fail, because there is an information flow from a high-level variable to a low-level variable in the assignment l := h. Even if at the very end the value of the variable l is always 0, most of the times, type-systems are not sensitive to such fact.

To overcome this gap, several approaches based on semantic models were proposed to prove information flow properties formalised as non-interference. In this thesis we rely on the *self-composition* technique proposed by Barthe et al. [17] to prove composition-based properties using off-the-shelf deductive verification tools. As introduced in Chapter 2, this technique relies on the composition of the program with its renamed copy to prove non-interference-like properties, using *Hoare logic* [54, 47] based systems. Some researchers studied how this technique can be applied to existing programming languages and tools. Next we describe some of this work.

Self-composition The work of Barthe et al. [17] was inspired by Leino and Joshi [57], the first to propose a semantic approach to check non-interference. Leino and Joshi identified several desirable features of this approach. It gives a more precise characterisation of security, it applies to all programming constructs whose semantics are well-defined and it can be used to reason about indirect information leakage through variations in program behaviour (e.g., whether or not the program terminates). The authors formalized non-interference as

$$\text{S is secure} \equiv (\text{HH;S;HH} \doteq \text{S;HH})$$

where S is a program and *HH* denotes the idea: “assign an arbitrary value to high variables”. Intuitively, the property expresses an equivalence between two program executions: (1) assigning an arbitrary value to high variables before executing the program, then executing the program and finally making the arbitrary assignment again; and (2) running the program and making the arbitrary assignment only once, right at the end. Informally, the property tries to capture the idea that the public outputs of the program are not affected by any confidential inputs.

Some attempts to model the property of Leino and Joshi in program logics using JML [62] were presented by Warnier and Oostdijk [95]. The authors proposed an algorithm, based on the strongest postcondition calculus [41], that from an input file written in a subset of sequential `Java`, generates an annotated source file with specification patterns for confidentiality in JML. Intuitively Leino and Joshi’s noninterference formalization is seen here as a property between program states: the values of low variables in the post-state are independent of the values of high variables in the pre-state.

Terauchi and Aiken [88] identified problems in the self-composition approach, arguing that automatic tools aren’t powerful enough to verify this property over programs of realistic size. To compensate for this, the authors proposed a program transformation technique to an extended version of the self-composition approach. This incorporates the notion of security levels downgrading through *relaxed non-interference* [63] property, to enforce secure information flow. The program transformation technique is based on rewriting the original program composed with its renamed version, but rather than replicating the original code, the renamed version is interleaved and partially merged with it. This transformation is assisted by type-directed translation rules.

Naumann [76] extended Terauchi and Aiken’s work to encompass heap objects, adding ghost fields to every object to express heap partition into dashed and undashed parts and to encode a partial bijection between those parts. This author also presented a systematic method to validate transformations on self-composition, proposed in [88], and reported his experiments of applying self-composition in object-oriented programs to prove noninterference with ESC/JAVA2 [56] and Spec # [15] tools.

Dufay et al. [42] proposed an extension of the JML specification to enforce noninterference through self-composition. The `Krakatoa` [67] tool, used to implement this extension, generates proof obligations for the Coq proof assistant, from JML-annotated `Java` programs. This extended annotation language allows a simple definition of noninterference in `Java` programs, although generated proof obligations are complex and in some methods they could not be completed. The authors appealed to *ghost* variables to keep track the invoked method parameters and result values of the first run to help the resolution of some proof obligations.

Beringer and Hofman [23] adapted the definition of self-composition to prove the absence of illicit flows in programs written in IMP. IMP is an imperative language equipped with an augmented type system that enforces termination-insensitive noninterference. The self-composition definition in this particular case is based on an invariant property ϕ which intuitively tries to relate the pre-states and post-states with the intermediate states achieved when a program is self-composed with itself. The authors also expressed this property to cover arbitrary security lattices and showed that from the typing derivations, it is possible to automatically generate proofs in programs logic.

In our work we extend the notion self-composition to account for a wide range of security properties that can be proved using composition-based approaches. However, contrary to Terauchi and Aiken [88], instead of modifying the target program to use off-the-shelf verification tools, we propose a methodology to increase the level of automation of the proof process. This methodology presents some similarities with the work of Beringer and Hofman [23] as we will see in Chapter 4.

Downgrading information flow policies Although the standard definition of noninterference seems the most appropriated to model information flow policies, in some cases it is too restrictive. Sometimes, it is not feasible to demand for full noninterference. For example, a program that checks a password always releases some information about confidential information, and would therefore be rejected under a strict noninterference definition. Significant work has been done to refine the noninterference definition to allow the downgrading of the security level associated with particular pieces of information [3, 64, 74, 75, 73, 49, 72, 91]. Sabelfeld and Sands [81] present a comprehensive survey about the directions of information downgrading. The authors proposed a classification of the basic goals that underlie declassification according to *what* can be downgraded, *who* can downgrade the information, *when* information can

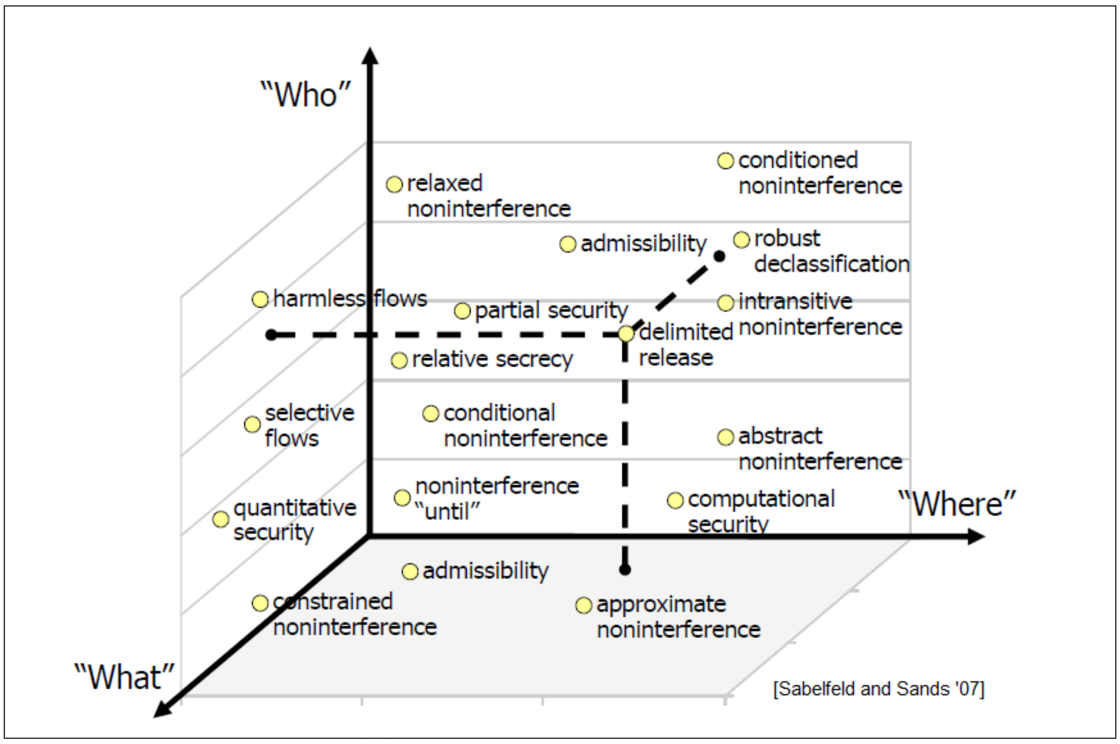

Figure 3.2: Dimensions of information release

be downgraded and *where* the system can release information. Figure 3.2 [81] provides a visual road-map of the main directions of current research on this topic.

Relaxed non-interference, proposed by Li and Zdancewic [63], is an example of a refinement of noninterference concerned about *how* information can be declassified. Their framework relies on type-checking and the end-user is free to specify in which circumstances the information can be downgraded. Other techniques such as the *decentralized label model* [72, 74, 73, 90, 101, 91, 71], *robust declassification* [100, 99, 75, 28], *abstract non-interference* [49], *intransitive noninterference* [64], and *dynamic labels* [102], are also examples of different models which rely on relaxed notions of non-interference.

Although in the thesis we do not directly address this subject, we believe that some of the techniques we propose can be used to capture some of the relaxed notions of non-interference proposed in the literature. Nevertheless, we leave this for future work. Similarly, significant work has been done in extending notions of noninterference, to programs that are not deterministic, which we do not consider in this work and also leave as future work.

3.5 Functional correctness

Functional correctness properties are related with verifying the correctness of the functional requirements in software systems, i.e., if they work as prescribed. This verification is based in some intended specification of the systems behavior. The goal is to establish if the input-output behaviour of the implementation matches that of the specification.

The specification of a cryptographic algorithm is often given as a reference implementation. This is the case, not only in symmetric-key techniques such as ciphers, message authentication codes and cryptographic hash functions; but also in the implementation of algebraic calculations supporting public-key techniques such as digital signature and encryption schemes. When producing an implementation, the programmer will follow the operational description but is free to introduce optimisations or internal reorganisations, say for improving efficiency or maintainability, or satisfying non-functional security properties, as long as the input-output behaviour is preserved.

To some extent, the specification acts as a reference implementation and verifying functional correctness reduces to proving program equivalence. Indeed, the sort of equivalence proof we require for cryptographic software configures what in software engineering is usually known as *code refactoring*. Software code refactoring aims to introduce optimisations in the code by changing its internal structure, but maintaining its observable/external behaviour. Functional correctness of cryptographic implementations is then a proof of equivalence between a reference implementation and an optimised version of the code. In Chapter 4 we propose a methodology to verify such properties.

In [97], the authors aim to verify the functional correctness of the AES block cipher implementation, written in C. They first extract the original FIPS [1] specification of the algorithm in PVS [78], then translate the original implementation (written in C) into SPARK Ada [12] language and annotate it with pre- and postconditions. Next the authors apply a semantic-preserving set of refactoring steps to that specification. Afterwards, from the last refactored version they mechanically extract a high-level specification (in PVS), using a tool called *Echo* [86]. To verify if the original implementation satisfies the PVS specification, they check if the high-level specification (extracted from the refactored version of the program) implies the original specification (extracted from the FIPS specification). We remark that, although it has the same purpose, our approach distinguishes itself from the one presented in [97].

A continuation of this work is presented in [96], where the authors explore the notion of *semantics preserving* refactoring transformations. The idea is that a refactoring transformation of a program P to a program P' is semantic preserving if for the same initial states they terminate in the same final states. Our definition of *equivalence by composition* (introduced in Chapter 2) aims to prove exactly this property, however our technique is directly applied to the original (C) implementations of the algorithms.

3.6 Side-channel countermeasures

A lot of research has been done to verify if the software systems are secure against side-channels attacks. In Chapter 5 we give our contribution in this field, by proposing a technique to verify if cryptographic implementations are secure against a specific set of side-channel attacks.

Volpano and Smith [94] first explored the use of type systems to protect programs against covert termination and timing channels. We apply deductive verification techniques to this end, yet we do not turn away completely from their work. Specifically, the authors define a *timing agreement* theorem which refers to a type-system that essentially captures our notion of security. The distinction between both security notions relies on the fact that our definition, being defined by semantic means, is slightly more inclusive. But admittedly, our main motivation for departing from the type-based approach was methodological, since we want to rely on the same set of deductive tools used in the verification of other properties (e.g. information flow properties).

The *Program Counter security model* (PC-model) proposed by D. Molnar et. al [68] captures the behavior of an attacker capable of observing the sequence of program counter positions during the execution of programs. Our security definition also takes into account these sequences, however it is broader than the one proposed by D. Molnar et. al [68], since it also captures the memory accesses done by the program during its execution. The primary aim in [68] was not to check conformance of programs with a security property, but rather transform potentially insecure programs into secure ones. In particular, the authors were able to justify several established countermeasures found in the literature.

Svenningsson and Sands [87] have adopted the PC-model and addressed control-flow independence using self-composition. They also considered the issue of declassification, enabling the formal verification that only controlled amounts of leakage can occur (e.g. the leakage of the hamming weight of a secret during a modular exponentiation).

Regarding the security notions, our work differs from this in two main aspects. On one hand, we consider a more restrictive security notion where we also check for data memory access pattern independence. On the other hand, we do not consider declassification. Our approach to applying self-composition to a transformed version of the original program is close to [87]. However, not only do we present a full theoretical framework to justify our approach, but also and most importantly, our practical implementation approach allows us to go beyond the results reported in [87]. In particular, we have not restricted the class of accepted programs to the so-called *unnested* programs.

The side-channel related security policies we have addressed in our work can also be seen as integrity-preserving information-flow restrictions. Indeed, it is well known that one can see high variables as untrusted inputs, that (one wants to check) do not interfere with the control flow and addresses accessed by the program. Intuitively, one is showing that attackers manipulating these inputs cannot influence the behavior of the program. This sort of security policy is sometimes addressed through so-called *taint-analysis*. Static taint analysis techniques tend to be based on type systems [26] or on control-dependency graphs (CFG) [27]. Our work can be seen as an alternative approach to taint analysis.

Part I

Verifying security properties in cryptographic software

Chapter 4

Verifying functional correctness and information flow properties

As introduced in Chapter 1, the first part of this thesis is devoted to the application of deductive verification techniques to prove that cryptographic implementations enforce a specific set of security properties. In this chapter we focus on the formalisation and verification of information flow and functional correctness properties in cryptographic implementations, using the Frama-c framework. We apply deductive verification techniques to formally verify correctness with respect to reference implementations and absence of error propagation in cryptographic implementations. Furthermore, we show how Frama-c can be used to prove safety properties in such implementations. A case-study presents the application of such techniques to the openSSL implementation of the RC4 algorithm.

4.1 Verifying safety properties using Frama-c

Programming languages like C and C++ support arbitrary pointer arithmetic, casts and memory allocation and deallocation. Mismanagement of pointers, casts to incorrect (smaller) types or attempt to access arrays with invalid indexes, can cause memory errors during the program execution, forcing the program to stop. For example, *buffer overflow* may appear in programs that write more data into a buffer (memory location) than the memory that is allocated to that buffer, causing the corruption of adjacent data on the stack. The following program is an example of a C program that contains a buffer overflow.

```
#include <stdio.h>
#include <string.h>

int main(int argc, char *args[]) {
     char buff[100], copy[100];

     gets(buff);
     strcpy(copy, buff);
     return 0;
}
```

This program simply reads a string from the input and copies it to a buffer. The problem is that it does not prevent the users from writing a string with more than 100 characters. In fact, the function `strcpy` itself is not safe, because it does not guard against copying strings where the destination string is smaller than the origin. Thus, malicious users can take advantage of such exploit by choosing a string large enough, and changing the return address of the `strcpy` function to an address which contains code controlled by them. A way to prevent this kind of attacks is to guarantee that the program is (memory) safe. *Safety* here is related with preventing runtime errors which are due to not accounted situations in the evaluation semantics of the programming language. As introduced in Chapter 2 this can be verified using deductive verification techniques.

The Jessie plug-in in the Frama-c framework allows users to perform a safety analysis of the C code, which may be run independently of functional verification. This produces a special class of verification conditions, called *safety conditions*. Their validity implies that the program will execute safely with respect to a restricted set of common programming errors that may result in incorrect or unreliable implementations, or even to security vulnerabilities. These comprise memory safety, including the absence of buffer overflows, and also the absence of numeric errors due to overflows in integer calculations. We present next a case-study of verification of safety properties of cryptographic algorithms using the Frama-c framework.

Case Study: openSSL implementation of RC4

The implementation of a cryptographic algorithm chosen as a case study was the implementation of the RC4 algorithm in the openSSL library. RC4 is a symmetric cipher designed by Ron Rivest at RSA labs in 1987. It is a proprietary algorithm, and its definition was never officially released. Source code that allegedly implements the RC4 cipher was leaked on the Internet in 1994, and this is commonly known as ARC4

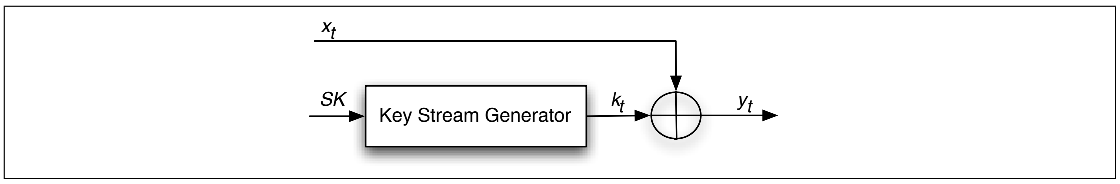

Figure 4.1: Block diagram of the RC4 cipher

due to trademark restrictions. In this work we will use the RC4 denomination to denote the definition adopted in literature [84].

RC4 selection was mostly inspired by the fact that this algorithm is widely used in commercial products, as it is included as one of the recommended encryption schemes in standards such as TLS, WEP and WPA. In particular, the selected implementation of RC4 provided in open-source library openSSL is pervasively used.

RC4 algorithm In cryptographic terms, RC4 is a synchronous stream cipher, which means that it is structured as two independent blocks, as shown in Figure 4.1. The security of the RC4 cipher resides in the strength of the key stream generator, which is initialized with a secret key SK. The key stream output is a byte sequence k_t that approximates a perfectly random bit string, and is independent of plaintext and ciphertext (we adopt the most widely used version of RC4, implemented in C programming language, which operates over byte-sized words). The encryption operation consists simply of XOR-ing each plaintext byte x_t with a fresh key stream byte k_t. Decryption operates in an identical way.

The key stream generator operates over a state which includes a permutation table $S = (S[l])_{l=0}^{l=255}$ of (unsigned) byte-sized values, and two (unsigned) byte-sized indices i and j. We denote the values of these variables at time t by S_t, i_t and j_t. The state and output of the key stream generator at time t (for $t \geq 1$) are calculated according to the following recurrence, in which all additions are carried out modulo 256.

$$i_t = i_{t-1} + 1 \tag{4.1}$$

$$j_t = j_{t-1} + S_{t-1}[i_t] \tag{4.2}$$

$$S_t[i_t] = S_{t-1}[j_t] \tag{4.3}$$

$$S_t[j_t] = S_{t-1}[i_t] \tag{4.4}$$

$$k_t = S_t[S_t[i_t] + S_t[j_t]] \tag{4.5}$$

The initial values of the indices i_0 and j_0 are set to 0, and the initial value of the permutation table S_0 is derived from the secret key SK. The details of this initialisation are immaterial for the purpose of this thesis, as they are excluded from the analysis.

In Appendix A.1 we include the `C` implementation of RC4 from openSSL open-source library. The function receives the current state of the RC4 key stream generator (`key`), and two arrays whose length is provided in parameter `len`. The first array contains the plaintext (`indata`), and the second array will be used to return the ciphertext (`outdata`). The same function can be used for decryption by providing the ciphertext in the `indata` buffer. Notice that this implementation is much less readable than the concise description provided above, as it has been optimised for speed using various tricks, including macro inlining and loop unrolling.

Safety analysis In Appendix A.2 we enrich the original implementation (Appendix A.1) with annotations that facilitate the verification of a set of safety properties (memory safety and absence of numeric errors) supported by `Frama-c`. This annotated version of RC4 gave rise to 869 verification conditions. All of these verification conditions could be automatically proved using a set of automatic theorem provers that includes Simplify, Alt-Ergo, and Z3. Important points in the verification process are described next.

Frama-c interprets C primitive types (e.g. `char`, `int`, etc.) as integers with different precisions, based on the number of bits of each type. This means that a number of proof obligations must be automatically generated to ensure the validity of each arithmetic operation, by imposing range limits on the corresponding results. Proof obligations that ensure that every memory access is safe are also automatically generated. Note that, even though these proof obligations do not result from explicit assertions made by the programmer, it is usually necessary to annotate the code with preconditions that permit justifying the proof goals. These preconditions limit the analysis to function executions for which the caller has provided valid inputs. To better explain this principle, recall the inference system for safety-sensitive Hoare triples of the *While*^{C} language (Section 2). The rule referent to the command $\mathtt{a}[e_1] := e_2$ has the following safety condition:

$$\varphi \rightarrow (\mathsf{safe}(e_1) \wedge 0 \leq e_1 < \mathsf{len}(\mathsf{a}) \wedge \mathsf{safe}(e_2))$$

where φ is the precondition and $\mathsf{safe}(\mathsf{e})$ denotes that the evaluation of e must be safe, i.e. does not generate a run-time error (the condition $\mathsf{safe}(e_1) \wedge 0 \leq e_1 < \mathsf{len}(\mathsf{a})$ comes from the definition of $\mathsf{safe}(\mathtt{a}[e_1])$). The evaluation of the command $\mathtt{a}[e_1] := e_2$ in C language is very similar to that of *While*^{C}. However, if the array a is a function parameter passed by reference, there is no away to statically infer the length of a, which interferes with the proof of the safety condition $0 \leq e_1 < \mathsf{len}(\mathtt{a})$. One way to solve this problem is to establish as a precondition that the length of `a` is some natural **n** and the code consumer has to fulfill this contract in order to guarantee that his/her program is safe.

For instance, in RC4 one must impose that the `indata` and `outdata` arrays have a valid addressable range between `0` and `len-1` for the safety conditions to be valid. This is expressed by adding the following annotations.

```
...
@  \valid(indata + (0..(len-1))) &&
@  \valid(outdata + (0..(len-1)));
...
```

In Appendix A.2 it is visible that the required preconditions also include the validity of the memory region in which the RC4 key stream generator state (`key`) is passed to the function.

As introduced in Chapter 2, Frama-c considers that program termination is a safety property, and it automatically generates conditions to prove loop termination. Since the RC4 algorithm from Appendix A.2 includes two loops, one has to annotate the program with loop variants, to be able to prove all the generated safety conditions. Recall that these are program expressions whose value decreases with each iteration, and are included in the program as annotations. For example, in the following code of the first loop, the variant clause expresses that at each iteration the value of the index `i` tends to `0`, and the invariant permits establishing that index `i` lies between `0` and `len»3L`.

```
 ...
/*@ loop invariant (0 < i <= (len>>3L)) &&
 @      indata == old_indata + ((old_i   i)*8) &&
 @      outdata == old_outdata + ((old_i   i)*8);
 @ loop variant i;
 @*/
while(1)
 {
 RC4_LOOP(indata,outdata,0);
...
```

To keep track of how the `indata` and `outdata` pointer values change during the loop execution, the invariant also includes annotations which relate the initial values (before loop execution) of the pointers and their values at each iteration.

Given the high number of proof obligations to be proved, and to guide the automatic provers in the process of establishing the validity of some of these conditions, additional assertions were introduced in the code. For example, at the end of the first loop, one assertion is introduced to *force* the provers to pinpoint the condition that must be valid

```
int fib1(int n) {
  int f1 = 0; int f2 = 1; int tmp = 0; int i=0;
  while (n > 0) { tmp = f1 + f2; f1 = f2; f2 = tmp; n = n - 1; }
  return f1;
}
```

Listing 4.1: Reference implementation of the Fibonacci algorithm in C

```
int fib2(int n) {
   int f1 = 0; int f2 = 1;
   while (n > 0) { f2 = f1 + f2; f1 = f2 - f1; n = n - 1; }
   return f1;
}
```

Listing 4.2: Optimised version of the implementation of the Fibonacci algorithm in C

at the end of the loop execution. Whenever a proof obligation is proved, it becomes part of the context and can be used to prove subsequent goals.

Finally, and given that cryptographic code tends to make use of some arithmetic operators that are not commonly used in other application domains, we noted that the proof tools lacked appropriate support in some cases, namely for bit-wise operators. To overcome this difficulty we added some very simple axioms to the annotated RC4 code that express bounds on the outputs of these operators.

4.2 Proofs by composition and self-composition

In this section we introduce a methodology to derive functional correctness proofs using the *equivalence by composition* technique introduced in Chapter 2. As an example of the application of this technique consider two versions of the implementation of the algorithm to calculate the n^{th} element of the *Fibonacci* sequence, written in C and shown in Listings 4.1 and 4.2. Note that, the observable behaviour of both implementations is the same: the input of both implementations is a natural number n and the result should be the n^{th} element of the Fibonacci sequence. The *refactoring step* only removes the need to use the variable tmp.

Consider that we want to prove the correctness of the code presented in Listing 4.2 with respect to reference implementation presented in Listings 4.1. Applying the *equivalence by composition* technique we obtain the program depicted in Listing 4.3. Firstly,

```
/*@ requires n==n1;
  @ ensures *fib1 == *fib2;
  @*/
void fib_composed(int n, int n1, int* fib1, int* fib2) {
      int f1 = 0; int f2 = 1; int tmp = 0;
      int f11 = 0; int f21 = 1;

      while (n > 0) { tmp = f1 + f2; f1 = f2; f2 = tmp; n = n - 1; }
      *fib1 = f1;

      while (n1 > 0) { f21 = f11 + f21; f11 = f21 - f11; n1 = n1 - 1; }
      *fib2 = f11;
}
```

Listing 4.3: Recursive implementation of the Fibonacci algorithm in C

we compose the Listing 4.1 with the renamed copy of Listing 4.2 (every variable x is renamed to x_1) and the return values of both implementations are passed by reference in the variables `*fib1` and `*fib2` (these must be pointers, otherwise their value cannot be captured in the post-state). Then, using the ACSL notation, we establish as a precondition that all input values are equal (`requires n==n1`) and as a postcondition that all output values must be equal too (`ensures *fib1==*fib2`).

Because both implementations contain loops, one has to annotate each program with a loop invariant to prove the postcondition. Sometimes it is hard to find the appropriate loop invariant which exactly captures the state transformation introduced in the loop evaluation. For this reason, we propose a methodology, based on the notion of *natural invariant*, to automate the definition of the loop invariant and consequently the proof process. This methodology can be summarised as follows:

Loop invariants predicates First, extract a specification of a program from its relational semantics. Since the critical point of the verification process is the automatic construction of appropriate loop invariants, each invariant is turned into a predicate, used to annotate the respective loop in the source code.

Lemmas Second, identify and interactively prove additional facts involving the named invariant predicates. For equivalence proofs, for example, these facts correspond to basic *refactoring steps* that are recurrently used in the development of cryptographic software. They are written as lemmas that capture the non-trivial parts of the proofs required for verification.

Proof Finally, augment the source file with the previous lemmas, which can be justified once-and-for-all by interactive proofs. The availability of these lemmas will allow automatic provers to carry out the verification process, validating the verification conditions generated by a potentially large number of composition-based proofs.

When both programs share much of the underlying control structure, the user may easily guide the interactive verification process by providing hints on the relevant code refactorings. The remaining parts can be checked with a high degree of automation.

4.2.1 Natural invariants

The notion of *natural invariant* arises from the difficulty of applying the *self-composition* technique. Note that the generalization of self-composition leads to similar difficulties to those mentioned for standard self-composition [17]. The difficulties pointed out by Terauchi and Aiken [88], are related with proving the postcondition in programs with loops. After a loop a new state is always introduced in the program, and even if the loop does not assign any value to any variable, a final state for each variable is introduced [1]. Consequently, the postcondition goals cannot be proved because no information is present concerning the values of the variables in the final state. To show how the values of the variables change during loop execution, a loop invariant must be introduced. Defining an appropriate loop invariant that captures the change of the state, can be a non-trivial task even for simple examples. Our methodology is based on the definition of a *natural invariant* which consists in the extraction of the relational specification from each program, such that the program trivially satisfies it.

Informally, this methodology resides on the definition of an abstract invariant that captures in logical form the state transformation associated with the loop, written as a formula that uses an inductively defined predicate (as supported by Frama-c). Let $\vec{v}$ denote the vector of variables used inside a given loop. The predicate

$$natinv(\vec{v_i}, \vec{v})$$

will be defined with the meaning that the execution of the loop started with initial values of the variables given by $\vec{v_i}$; and the current values of the variables are given by $\vec{v}$. The inductive definition of this predicate has in general a *base case* of the form

$$natinv(\vec{v_i}, \vec{v_i})$$

[1]Recent versions of Frama-c eliminate the need to create new states for variables which are not assigned to any value during the loop execution.

(corresponding to the loop initialization) and an *inductive case* of the form

$$natinv(\vec{v_i},\vec{v_t}) \rightarrow B \rightarrow R(\vec{v_t},\vec{v}) \rightarrow natinv(\vec{v_i},\vec{v})$$

where B is the boolean condition of the loop, and the formula $R(\vec{v_t},\vec{v})$ relates the values of the variables in two successive iterations.

Natural invariants can be used to automate the proof process of general composition-based proofs. Their formal definition is presented next.

Relational specification Consider safe $While^C$ (toy language defined in Chapter 2) programs, i.e., $While^C$ programs whose safety were previously verified. For states s and s' we define:

$$\begin{aligned}
\mathsf{spec}_{\mathbf{skip}}(s,s') &= s \equiv s' \\
\mathsf{spec}_{C_1;C_2}(s,s') &= \exists s''.\ \mathsf{spec}_{C_1}(s,s'') \wedge \mathsf{spec}_{C_2}(s'',s') \\
\mathsf{spec}_{\mathtt{x:=}e}(s,s') &= s' \equiv s[\mathtt{x} \leftarrow [\![e]\!](s)] \\
\mathsf{spec}_{\mathsf{a}[e_1]\mathtt{:=}e_2}(s,s') &= s' \equiv s[\mathsf{a} \leftarrow \mathsf{upd}(\mathsf{a},[\![e_1]\!](s),[\![e_2]\!](s))] \\
\mathsf{spec}_{\mathbf{if}\ (e)\ \{C_1\}\ \mathbf{else}\ \{C_2\}}(s,s') &= (([\![e]\!](s) = \mathbf{true}) \wedge \mathsf{spec}_{C_1}(s,s')) \vee \\
&\quad\ (([\![e]\!](s) = \mathbf{false}) \wedge \mathsf{spec}_{C_2}(s,s')) \\
\mathsf{spec}_{\mathbf{while}\ (e)\ C}(s,s') &= \exists n.\ \mathsf{loop}^n_{e,\mathsf{spec}_{C\restriction_R}(s,s''),\mathsf{spec}_{C\restriction_K}(s'',s')}(s,s') \ \wedge\ ([\![e]\!](s') = \mathbf{false})
\end{aligned}$$

where

- $C\restriction_K$ denotes the restriction of C to the statements responsible for the control flow of the loop (loop condition);
- $C\restriction_R$ the restriction of C to the remaining instructions (i.e., statements which do not influence the control flow of the loop) – note that this can be obtained by a simple dependency analysis[2].

The relation $\mathsf{loop}^n_{B,R,K}(s,s')$ denotes the loop specification for the body R and increment K under condition B and is inductively defined by

[2]Observe that $\mathsf{spec}_{C\restriction_R}(s,s'') \wedge \mathsf{spec}_{C\restriction_K}(s'',s') \equiv \mathsf{spec}_C(s,s')$, because $C\restriction_R$ and $C\restriction_K$ write on disjoint parts of the state. Besides, $C\restriction_K$ also reads from a disjoint part of the state.

$$\mathsf{loop}^0_{B,R,K}(s,s') = s \equiv s'$$
$$\mathsf{loop}^{\mathsf{succ}(n)}_{B,R,K}(s,s') = \exists s''\ s'''.\ \mathsf{loop}^n_{B,R,K}(s,s'') \wedge ([\![B]\!](s'') = \mathbf{true}) \wedge R(s'',s''') \wedge K(s''',s')$$

where $\mathsf{succ}(n)$ denotes the successor of the natural n. Let π^B the project of the fragment of the state that influences the control flow structure (i.e., loop conditions). We remark that the relation R enjoys the following property on states: $\forall s\ s'.\ R(s,s') \Longrightarrow \pi^B(s) \equiv \pi^B(s')$.

The relation $\mathsf{loop}^n_{B,R,K}(s,s')$ provides a natural choice for a loop's invariant; and this is the reason to call it the *natural invariant* for the loop. The following example demonstrates how one can easily extract the relational specification of a simple program with loops and the corresponding natural invariant.

Example 1. *Consider the program P defined as:*

```
i := 0;
while(i < n) {
  x := x + y;
  i := i + 1;
}
```

and let L denote the code of the while loop of P and B its body. The relational specification of P is given by:

$$\mathsf{spec}_P(s,s') = \exists s''.\ \mathsf{spec}_{i:=0}(s,s'') \wedge \mathsf{spec}_L(s'',s')$$
$$= \exists s''. s'' \equiv s[i \leftarrow 0] \wedge (\exists n.\ \mathsf{loop}^n_{i<n,R(s'',s'''),K(s''',s')}(s'',s') \wedge [\![i<n]\!](s') = \mathbf{false})$$

where $R(s,s) = \mathsf{spec}_{B\restriction_R}(s,s')$ *and* $K(s,s') = \mathsf{spec}_{B\restriction_K}(s,s')$*. The predicate* $\mathsf{loop}^n_{i<n,R,K}(s,s'')$ *is defined as:*

$$\mathsf{loop}^0_{i<n,R(s,s''),K(s'',s')}(s,s') = s \equiv s'$$
$$\mathsf{loop}^{\mathsf{succ}(n)}_{i<n,R(s_1,s'),K(s',s_5)}(s_1,s_5) = \exists s_2 s_3 s_4.\ \mathsf{loop}^n_{i<n,R(s_1,s_2),K(s_2,s_3)}(s_1,s_3) \wedge$$
$$([\![i<n]\!](s_3) = \mathbf{true}) \wedge R(s_3,s_4) \ \wedge\ K(s_4,s_5)$$

and

$$R(s,s') = \mathsf{spec}_{B\restriction_R}(s,s') = \mathsf{spec}_{x:=x+y}(s,s') = s' \equiv s[x \leftarrow [\![x+y]\!](s)]$$
$$K(s,s') = \mathsf{spec}_{B\restriction_K}(s,s') = \mathsf{spec}_{i:=i+1}(s,s') = s' \equiv s[i \leftarrow [\![i+1]\!](s)]$$

□

The definition makes explicit the *iteration rank* (iteration count) in superscript – we will see, further in this chapter, that this will often be convenient in the proofs (when omitted, it should be considered as existentially quantified). Subscripts will be omitted (both in loop and spec) when the corresponding programs are clear from the context.

In practice, it is convenient to produce the relational specification formula in *prenex-form*. This is easily accommodated introducing new fresh state variables in each elementary statement:

$$\mathsf{spec}_{C_1;C_2;\ldots;C_n}(s, s') = \exists s_0 \ldots s_n,\ s_0 = s \wedge s_1 = P_{C_1}(s_0) \wedge \ldots \wedge s_n = P_{C_n}(s_{n-1}) \wedge s' = s_n$$

where $s_i = P_{C_i}(s_{i-1})$ denotes $\mathsf{spec}_{C_i}(s_{i-1}, s_i)$, i.e., corresponds to the atomic specification associated with statement C_i. By construction, spec enjoys the following properties.

Lemma 12. *Let R and K be deterministic relations on states and B a boolean condition. Then,* $\mathsf{loop}_{B,R,K}(s, s')$ *is deterministic whenever* $[\![B]\!](s') = \textbf{\textit{false}}$*, i.e.*

loop synchronisation $\forall n_1\ n_2\ s_1\ s_2\ s_1'\ s_2'.\ s_1 \equiv s_2 \wedge \mathsf{loop}^{n_1}_{B,R,K}(s_1, s_1') \wedge$
$([\![B]\!](s_1') = \textbf{\textit{false}}) \wedge \mathsf{loop}^{n_2}_{B,R,K}(s_2, s_2') \ \wedge\ ([\![B]\!](s_2') = \textbf{\textit{false}}) \Longrightarrow n_1 = n_2;$

loop determinism $\forall n\ s_1\ s_2\ s_1'\ s_2'.\ s_1 \equiv s_2 \wedge \mathsf{loop}^{n}_{B,R,K}(s_1, s_1') \wedge$
$\mathsf{loop}^{n}_{B,R,K}(s_2, s_2') \Longrightarrow s_1' \equiv s_2'.$

Proof. Both statements are proved by a simple induction (on $\max(n_1, n_2)$ in the first case, and n in the second). We remark that the proof of the synchronisation lemma requires the proof of the following property:

$$\forall n_1\ n_2\ s_1\ s_2\ s_1'\ s_2'.\ s_1 \equiv s_2\ \wedge\ \mathsf{loop}^{n_1}_{B,R,K}(s_1, s_1') \wedge \mathsf{loop}^{n_2}_{B,R,K}(s_2, s_2')\ \wedge\ n_1 < n_2 \Longrightarrow ([\![B]\!](s_1') = \textbf{true})$$

that can be easily proved by simple induction on n_2. □

Proposition 13. spec *is a morphism that preserves* $\equiv$ *and is deterministic. More precisely, for every program fragment C and states* s_1, s_2, s_1', s_2',

- *If* $s_1 \equiv s_2$, $s_1' \equiv s_2'$ *and* $\mathsf{spec}_C(s_1, s_1')$ *then* $\mathsf{spec}_C(s_2, s_2')$.
- *If* $\mathsf{spec}_C(s, s_1')$ *and* $\mathsf{spec}_C(s, s_2')$ *then* $s_1' \equiv s_2'$.

Proof. By induction on the structure of P using Lemma 12. □

The strategy for reasoning about multiple executions (for self-composition or to justify interesting refactorings) is based on identifying a set of general lemmas that can be proven once-and-for-all, and then included in the annotations provided to the verification platform, allowing other proof obligations to be automatically discharged.

4.2.2 Self-composition lemmas

The determinism property is not relevant to reason about a non-interference property by self-composition: it merely states that the two instances of the program will produce the same outputs when all of their inputs are equal. What is needed is a rephrasing of that property using an equality relation on low-security variables. If the control structure of the program does not depend on high-security variables, the determinism property proof can be carried over to non-interference lemmas. More explicitly, Lemma 12 can be reformulated as follows.

Lemma 14. *Let B be a boolean condition and R and K deterministic relations on the low part of the states, i.e.,*

$$\forall s_1, s_2, s'_1, s'_2,\ \ s_1 \equiv_L s_2 \wedge R(s_1, s'_1) \wedge R(s_2, s'_2) \Rightarrow s'_1 \equiv_L s'_2;$$
$$\forall s_1, s_2, s'_1, s'_2,\ \ s_1 \equiv_L s_2 \wedge K(s_1, s'_1) \wedge K(s_2, s'_2) \Rightarrow s'_1 \equiv_L s'_2.$$

Then, $\mathsf{loop}_{B,R,K}(s, s')$ *is deterministic on the low part of the states whenever* $[\![B]\!](s') =$ **false***, i.e.*

loop synchronisation $\forall n_1\ n_2\ s_1\ s_2\ s'_1\ s'_2.\ s_1 \equiv_L s_2 \wedge \mathsf{loop}^{n_1}_{B,R,K}(s_1, s'_1) \wedge$
$([\![B]\!](s'_1) = \textit{\textbf{false}}) \wedge \mathsf{loop}^{n_2}_{B,R,K}(s_2, s'_2) \ \wedge\ ([\![B]\!](s'_2) = \textit{\textbf{false}}) \Longrightarrow n_1 = n_2;$

loop determinism $\forall n\ s_1\ s_2\ s'_1\ s'_2.\ s_1 \equiv_L s_2 \wedge \mathsf{loop}^{n}_{B,R,K}(s_1, s'_1) \wedge$
$\mathsf{loop}^{n}_{B,R,K}(s_2, s'_2) \Longrightarrow s'_1 \equiv_L s'_2.$

Proof. Both statements are proved by a simple induction (on $\max(n_1, n_2)$ in the first case, and n in the second). □

This lemma establishes that a non-interference result for each loop follows easily from noninterference in its body. Its precondition can be seen as an additional proof-obligation that must be verified. For simplicity, to prove noninterference of a program, instead of annotating the program with the properties specified in Lemma 14, we only provide one lemma which directly results from the ones above.

Lemma 15. *(Self-composition lemma)*

Let B be a boolean condition, $R(s,s')$ and $K(s,s')$ deterministic relations on the low part of the states.

$$\forall s_1, s_2, s'_1, s'_2.\ s_1 \equiv_L s_2 \ \wedge\ \mathsf{loop}^{n_1}_{B,R,K}(s_1, s'_1) \wedge ([\![B]\!](s'_1) = \mathbf{false}) \wedge \mathsf{loop}^{n_2}_{B,R,K}(s_2, s'_2) \wedge ([\![B]\!](s'_2) = \mathbf{false}) \Rightarrow s'_1 \equiv_L s'_2.$$

Proof. By induction on the number of iterations and using Lemma 14. □

A Coq library to mechanise these proofs was proposed in [7], however its development is not part of this thesis work. We remark that proving noninterference for loop-free programs by self-composition can be easily automated.

4.2.3 Equivalence by composition lemmas

In the setting of equivalence proofs, justifying code refactorings is slightly harder when the refactorings affect loops. For the sake of presentation, we restrict our attention to specifications obtained from single loops with loop-free bodies, i.e., we consider natural invariants of the form $\mathsf{loop}_{B,\mathsf{spec}_{C\uparrow_R},\mathsf{spec}_{C\uparrow_K}}(s,s')$ where C contains no loops. This is sufficient to cover the refactorings needed for the case-studies addressed in Section 4.4. We note however that *natural invariants* can be easily defined for loops whose body also contain loops (as we will see in Section 4.5).

The simplest loop refactoring that can be addressed using our technique is *loop unrolling*, which detaches instances of the loop body. This transformation is justified by the following property that results from direct inversion of the definition of loop:

$$\forall n\ n'\ s\ s',\ \mathsf{loop}^n(s,s') \wedge n' < n \implies \exists s'',\ \mathsf{loop}^{n'}(s,s'') \wedge \mathsf{loop}^{n-n'}(s'',s')$$

Simple transformations like these are in fact better handled directly at the annotation level, rather than through explicit lemmas. A single natural invariant can be defined for the original loop, and then used in the annotations of the unrolled loop, in order to establish the necessary relation between the execution of the two programs. Let us illustrate this by a small example that mimics an optimising transformation for the real-world example presented in Section 4.4.

Example 2. *Consider the program:*

```
i := 0;
while (i<N) { x := x + y; i := i + 1; }
```

To implement it, the programmer chooses to unfold two copies of the original loop , yielding

```
N2 := N/2;
i := 0;
if (i<N2) then {
  while (i<N2) { x := x + y; i := i + 1; };
  i:=0;
  while (i<N2) { x := x + y; i := i + 1; };
  if (2*N2 != N) then x := x + y else skip
}
else skip
```

To verify the equivalence between this implementation and the original program it suffices to identify each loop invariant in the second program as the following,

$$\mathsf{loop}^{n/2}((x, y, i)@\mathsf{Init}, (x, y, i))$$

where $s@\mathsf{Init}$ *evaluates the state* s *in the pre-state of the loop, and* loop *refers to the natural invariant of the loop in the first program. By providing the invariant, we are making explicit the correspondence between both loop executions.* □

This kind of guidance is reasonable to expect from someone intending to prove correctness of the target implementation. Alternatively, one could establish that both programs are equivalent using direct logical arguments, as will be now explained. This would be the only option for more complex refactorings.

General loop fusions To justify more significant code refactorings such as loop fusions (i.e. combining the bodies of two consecutive loops with the same control structure), we need to rely on an explicit lemma. Consider the equivalence between two consecutive loops (loops 1 and 2) and one single *fused* loop (loop 3). This is reminiscent of another real-world code refactoring that will occur in our case-study in Section 4.4.

Let us denote the natural invariants of these loops by $\mathsf{loop}_1, \mathsf{loop}_2$ and loop_3, respectively. Since we assume that all the loops share the same control structure (loop condition and associated state) [3], it is possible to prove *mixed* synchronisation lemmas as the following one.

Lemma 16. *(Mixed synchronisation lemma)*

[3]This means that $B_1 = B_2 = B_3$ and $K_1 = K_2 = K_3$; and loops share the same set of initialisation statements.

Let π^B be the projection of the fragment of the state that influences the control flow structure, B a boolean condition and R_1, R_2 and K deterministic relations on states.

$$\forall n_1\ n_2\ s_1\ s_2\ s_1'\ s_2'.\ \pi^B(s_1) \equiv \pi^B(s_2) \wedge \mathsf{loop}_1^{n_1}(s_1, s_1') \wedge ([\![B]\!](s_1') = \mathbf{false}) \wedge \\ \mathsf{loop}_2^{n_2}(s_2, s_2') \wedge ([\![B]\!](s_2') = \mathbf{false}) \Longrightarrow n_1 = n_2$$

and each $\mathsf{loop}_i^{n_i}$ denotes $\mathsf{loop}_{B,R_i,K}^{n_i}$, for $i = 1, 2$.

Proof. The proof is a straightforward generalisation of the single loop version. □

Now let init denote a deterministic relation responsible for the initialisation of the variables that intervene in the control flow of the loop, such that,

$$\forall s\ s'.\ \mathsf{init}(s, s') \Longrightarrow \overline{\pi^B}(s) \equiv \overline{\pi^B}(s')$$
$$\forall s_1\ s_2\ s_1'\ s_2'.\ \mathsf{init}(s_1, s_1') \wedge \mathsf{init}(s_2, s_2') \Longrightarrow \pi^B(s_1') \equiv \pi^B(s_2')$$

where $\overline{\pi^B}$ corresponds to projection of the fragment of the state that does not influence the control flow structure of the program. Let also R^k denote the relation R at iteration k, i.e., $R^k(s, s') = R(s, s')\ \wedge\ \pi^B(s) = \pi^B(s_k)$ where s_k is the state such that $\forall s.\ \exists s'.\ \mathsf{init}(s, s') \wedge K^k(s', s_k)$ and K^k denotes the relation K applied k times.

We are now in conditions to establish the following main lemma that can be used to justify the fusion refactoring.

Lemma 17. *(Fusion refactoring)*
Let B a boolean condition and R_1, R_2, R_3 and K deterministic relations on states.

$$(\mathsf{BodyFusion}(R_1, R_2, R_3) \wedge \mathsf{BodySwap}(R_1, R_2)) \Rightarrow \\ \forall n\ s_1\ s_2\ s_1'\ s_1''\ s_1^1\ s_1^2\ s_2''\ s_2'.\ s_1 \equiv s_2 \wedge \mathsf{init}(s_1, s_1^1) \wedge \mathsf{loop}_1^{n_1}(s_1^1, s_1'') \wedge \\ ([\![B]\!](s_1'') = \mathbf{false}) \wedge \mathsf{init}(s_1'', s_1^2) \wedge \mathsf{loop}_2^{n_2}(s_1^2, s_1') \wedge ([\![B]\!](s_1') = \mathbf{false}) \wedge \\ \mathsf{init}(s_2, s_2'') \wedge \mathsf{loop}_3^{n_3}(s_2'', s_2') \wedge ([\![B]\!](s_2') = \mathbf{false}) \Longrightarrow s_1' \equiv s_2'$$

where

$$\mathsf{BodyFusion}(R_1, R_2, R_3) = \forall\ s\ s'.\ \exists s''.\ R_3(s, s') \equiv R_1(s, s'') \wedge R_2(s'', s')$$
$$\mathsf{BodySwap}(R_1, R_2) = \forall k\ k'\ s\ s'\ s''.\ k' < k \Rightarrow \exists s'''. \\ R_1^k(s, s'') \wedge R_2^{k'}(s'', s') \equiv R_2^{k'}(s, s''') \wedge R_1^k(s''', s')$$

and each $\mathsf{loop}_i^{n_i}$ *denotes* $\mathsf{loop}_{B,R_i,K}^{n_i}$*, for* $i = 1, 2, 3$.

Proof. By induction on the number of iterations and using Lemma 16. □

BodyFusion establishes *fusion* for the corresponding body relations for each iteration k; and BodySwap establishes that iteration k of the first loop commutes with iteration k', for $k' < k$ of the second. This last is required to capture the *less-than* relation on iterations. These predicates denote simple properties concerning the loop bodies which, as was the case with the self-composition lemmas, are all non-recursive and can thus be regarded as additional proof-obligations, easily discharged by automatic provers.

As stated, the proof of the fusion lemma is done by induction on the number of iterations and using Lemma 16. We explain now the proof intuition, by explaining the need of the BodyFusion and BodySwap properties to prove the lemma.

First, applying the mixed synchronisation lemma we can conclude that $n_1 = n_2 = n_3$ – observe that the predicate init establish that $\pi^B(s_1^1) \equiv \pi^B(s_1^2)$ and applying Lemma 16 follows that $n_1 = n_2$; since $s_1 \equiv s_2$, in particular $\pi^B(s_1) \equiv \pi^B(s_2)$ and again by Lemma 16 we have $n_1 = n_3$.

Now, we need to prove that starting from equal initial states, executing loop_1 and then loop_2, will generate a final state that is equivalent to the one obtained by the execution of loop_3. Assuming that n is the number of iterations, this intuition can be expressed using the following diagram:

$$\begin{tikzcd}
s_1 \arrow[r, "R_1^1"] \arrow[ddrr, bend right, "R_3^1"'] & s_1^1 \arrow[r, "R_1^2"] & s_1^2 \arrow[r, "R_1^3"] & \cdots \arrow[r, "R_1^n"] & s_1'' \arrow[r, "R_2^1"] & s_1''^1 \arrow[r, "R_2^2"] & s_1''^2 \arrow[r, "R_2^3"] & \cdots \arrow[r, "R_2^n"] & s_1' \\
& & & & & & & & \\
& & s_2^1 \arrow[rr, "R_3^2"'] & & s_2^2 \arrow[rrr, "R_3^3"'] & & & \cdots \arrow[uur, bend right, "R_3^n"'] &
\end{tikzcd}$$

where $[\![B]\!](s_1'') = [\![B]\!](s_1') = \textbf{false}$, s_1^i and $s_1''^i$, for $i = 1 \ldots n-1$, are the intermediate states achieved during each loop iteration, and R_j^k denotes the loop body j at each iteration k, for $j = 1, 2, 3$ and $k = 1 \ldots n$.

The idea is that, unfolding each loop body n times, we need to prove that for each iteration k, the property $\forall s s'.\ R_3^k(s, s') \equiv \exists s''.\ R_1^k(s, s'') \wedge R_2^k(s'', s')$ holds (i.e., an instance of the BodyFusion predicate). Notice that this proof requires *reordering* each body execution for each iteration k. Therefore, a set of exchanges must be done between loop bodies of different iterations, which it is only possible if we are able to prove the BodySwap predicate.

For instance consider that $n = 3$. Unfolding the definitions of $\mathsf{loop}_i^3(s, s')$, for $i = 1, 2$ in $\mathsf{loop}_1^3(s_1, s_1'') \wedge \mathsf{loop}_2^3(s_1'', s_1')$ and assuming that $[\![B]\!](s) = \textbf{true}$ for any intermediate state s, we obtain:

$$\begin{tikzcd}
R_1^1 \arrow[r] & R_1^2 \arrow[r] & R_1^3 \arrow[r] & R_2^1 \arrow[r] & R_2^2 \arrow[r] & R_2^3
\end{tikzcd}$$

Now to prove the BodyFusion predicate for each iteration k one needs to reorder the loop body predicates for each iteration. So, starting with the swap,

$$\begin{tikzcd}
R_1^1 \arrow[r] & R_1^2 \arrow[r] & R_2^1 \arrow[r] \arrow[r, bend left, dashed] & R_1^3 \arrow[r] \arrow[l, bend left, dashed] & R_2^2 \arrow[r] & R_2^3
\end{tikzcd}$$

it is required to prove that $R_1^3 \cdot R_2^1 \equiv R_2^1 \cdot R_1^3$, i.e., an instance of the BodySwap predicate. If we apply this methodology to obtain the diagram,

$$\begin{tikzcd}
\mathsf{init} \arrow[r] & R_1^1 \arrow[r] & \mathsf{init} \arrow[r] & R_2^1 \arrow[r] & R_1^2 \arrow[r] & R_2^2 \arrow[r] & R_1^3 \arrow[r] & R_2^3
\end{tikzcd}$$

we have to prove $R_1^k \cdot R_2^{k'} \equiv R_2^{k'} \cdot R_1^k$, for each iteration k and k' such that $k' < k$. At the very end we have to use the BodyFusion predicate to prove the property $R_3^k \equiv R_2^k \cdot R_1^k$, for each iteration k.

4.2.4 Verification infrastructure

We have used the deductive verification tool included in the Frama-c framework (the Jessie plug-in), to do composition-based proofs in cryptographic software implementations. This section describes our use of this tool to support the approaches described above and provides an illustrative example.

Frama-c usage Frama-c takes as input annotated C programs in the form of ACSL files. In particular, loop invariants are mandatory for the verification to succeed. This means that in order to verify a property by composition, it is not enough to properly construct the composed program and to specify the intended contract (pre- and post-conditions) – this would certainly generate unprovable verification conditions. It is also required to complement the ACSL file with definitions and annotations. The following steps detail the procedure needed to perform the verification:

1. Extracting the relational specification of the code (which comprises the *natural invariants* definition of each loop);

2. Including ACSL definitions corresponding to the inductive properties associated to each loop;

3. For each loop specification, annotating the program with a loop invariant of the form

$$\mathsf{Inv}_{loop}(s) = \mathsf{loop}^{n}_{B,R,K}(s@\mathsf{Init}, s)$$

   where n is the number of iterations, B is the loop's condition and R and K the loop's body (K is the body fragment that influences the control structure and R corresponds to the remaining instructions), s@Init denotes the snapshot of the loop's initial state (Frama-c supports this notion through the use of explicit state labels in annotations, as we will see in Sections 4.3 and 4.4);

4. Augmenting the ACSL file with specific lemmas; these lemmas are important to allow the remaining proof goals, related with the composition-based proofs, to be discharged automatically.

5. Generating proof obligations using the deductive verification plug-in of the Frama-c framework (the Jessie plug-in);

6. Using an automatic prover (e.g. Alt-Ergo) to discard the generated obligations; and a proof assistant (e.g. Coq) to prove the composition-based lemmas.

The choice of the required lemma is based on the specific property under scrutiny (e.g. a self-composition lemma for a non-interference property). We remark that this user-dependent choice is an important ingredient for the success of the verification process. The goal of our method is to allow the user to concentrate on this critical part of the verification process by providing assistance in dealing with the remaining tasks, which are tedious but luckily prone to automation. Notice that in the case study presented in Section 4.4 the specification was extracted by hand, but we remark that the process is certainly amenable to mechanisation.

```
/*@ inductive fib2_loop(integer n1, integer n2, int f1i, int f1f,
 @                int f2i, int f2f){
 @   case base_case:
 @     \forall integer n;  \forall int f1,f2;  fib2_loop(n,n,f1,f1,f2,f2);
 @   case ind_case:
 @     \forall integer n1,n2,n3;  \forall int f1,f2,f11,f12,f21,f22;
 @     fib2_loop(n1,n2,f1,f11,f2,f21) ==> n2 > 0 ==>
 @     loop_body(n2,n3,f11,f12,f21,f22) ==> fib2_loop(n1,n3,f1,f12,f2,f22);
 @ }
 @*/
```

Listing 4.4: Definition of the natural invariant predicate

A simple example As an example, consider the implementations presented in Listings 4.1 and 4.2, which calculate the n^{th} element of the *Fibonacci* sequence. The first step to prove that the implementations are equivalent is to extract the relational specification of the code. According to the previous section, the specification for the code of Listing 4.2 (presented in the prenex-form as suggested) is given by:

$$\begin{aligned}\mathsf{spec}(x_1, x_2) = \exists s_0 \dots s_2,\ x_1 = s_0 \wedge s_1 = s_0[f_1 \leftarrow 0] \wedge s_2 = s_1[f_2 \leftarrow 1] \wedge \\ \exists n_0.\ \mathsf{loop}^{n_0}_{n>0,\mathsf{spec}_{C\restriction_R}(s_2,s_2'),\mathsf{spec}_{C\restriction_K}(s_2',s_3)}(s_2, s_3) \wedge ([\![n > 0]\!](s_3) = \textbf{false}) \wedge x_2 = s_3\end{aligned}$$

where $\mathsf{loop}^{n_0}_{n>0,R,K}(s, s')$ is the natural invariant defined by:

$$\begin{aligned}&\mathsf{loop}^{0}_{n>0,R,K}(s, s') = s \equiv s' \\ &\mathsf{loop}^{succ(k)}_{n>0,R,K}(s, s') = \exists s'' s'''.\ \mathsf{loop}^{k}_{n>0,R,K}(s, s'') \wedge ([\![n > 0]\!](s'') = \textbf{true} \wedge \\ &\qquad\qquad R(s'', s''') \wedge K(s''', s')\end{aligned}$$

and $R = \mathsf{spec}_{C\restriction_R}$ and $K = \mathsf{spec}_{C\restriction_K}$ are defined as:

$$\begin{aligned}\mathsf{spec}_{C\restriction_R}(s, s') &= \exists s_0 s_1.\ s_0 \equiv s[f_2 \leftarrow [\![f_1 + f_2]\!](s)] \wedge \\ &\qquad s_1 \equiv s_0[f_1 \leftarrow [\![f_2 - f_1]\!](s_0)] \wedge s_1 \equiv s' \\ \mathsf{spec}_{C\restriction_K}(s, s') &= s' \equiv s[n \leftarrow [\![n - 1]\!](s)]\end{aligned}$$

The definition of such invariant using the ACSL notation is shown in Listing 4.4, where loop_body corresponds to the predicate $\mathsf{spec}_{C\restriction_R} \cdot \mathsf{spec}_{C\restriction_K}$ defined in ACSL as follows:

```
/*@ predicate loop_body(integer n1, integer n2, int f11,
 @             int f12, int f21, int f22) =
 @             n2 == n1 - 1 && f22 == f11 + f21 && f12 == f22 - f11; @*/
```

```
/*@ requires n>=0 && n==n1;
  @ ensures *fib1 == *fib2;
  @*/
void fib_composed(int n, int n1, int* fib1, int* fib2) {
  int f1 = 0; int f2 = 1; int tmp = 0;
  int f11 = 0; int f21 = 1;

  //@ghost int n_old = n;
  //@ghost int n1_old = n1;

  //@ ghost goto L1;
  //@ ghost L1:

  /*@ loop invariant fib1_loop(n_old,n,tmp,\at(f1,L1),f1,\at(f2,L1),f2);
    @*/
  while (n > 0) { tmp = f1 + f2; f1 = f2; f2 = tmp; n = n-1; }
  *fib1 = f1;

  /*@ loop invariant fib2_loop(n1_old,n1,\at(f11,L1),f11,\at(f21,L1),f21);
    @*/
  while (n1 > 0) { f21 = f11 + f21; f11 = f21 - f11; n1 = n1-1; }
  *fib2 = f11;
}
```

Listing 4.5: Recursive implementation of the Fibonacci algorithm in C

In ACSL notation, (for this particular case) the values of successive iterations are represented by variables with different names, to capture the state transformation (in the following sections we will see that these values will be captured in ACSL in a different way). For example, `f11` represents the value of the variable `f1` in the previous state and `f12` represents its value in the current state. The boolean condition of the loop is $n > 0$. Thus, in the definition of the natural invariant, fib2_loop, the boolean condition that must be valid in the current iteration is `n2>0`, because `n2` represents the value of variable `n` in the previous state.

After defining the appropriate loop invariants, one has to annotate the loops with such predicates in the composed program, as shown in Listing 4.5. The natural invariant for the first loop is given by the predicate fib1_loop(integer i1, integer i2, int tmp, int f1i, int f1, int f2i, int f2).

So far, we are only able to discharge the proof obligations related with the loop invariants (initialisation and preservation). To prove the postcondition we need to include the following lemma:

```
/*@ lemma fusion:
  @ \forall int n_old,n,tmp,f1i,f1,f2i,f2,n1_old,n1,f11i,f11,f21i,f21;
  @ n_old==n1_old ==> f1i == f11i ==> f2i == f21i ==>
  @ fib1_loop(n_old,n,tmp,f1i,f1,f2i,f2) ==> n<=0 ==>
  @ fib2_loop(n1_old,n1,f11i,f11,f21i,f21) ==> n1<=0 ==> (f1 == f11 && f2 == f21);@*/
```

which captures the equivalence property, i.e., for equal initial states, the execution of both loops must produce the equal final states. Summing up, the resulting annotated input file includes:

- the definition of natural invariants of each loop;
- the equivalence lemma;
- loop invariants annotated with the natural invariant predicates;
- and pre- and postconditions annotations to express the desired property.

Afterwords, the Jessie plug-in can be used to automatically discharge all the proof obligations. We detail next how the proof of the equivalence lemma can be manually done. Alternatively, one can use the Coq library [7] developed for that purpose.

Let C_1 denote the loop body of Listing 4.1 and C_2 the loop body of Listing 4.2. Now, let R_1 denote the predicate $\mathsf{spec}_{C_1\upharpoonright_R}$, R_2 the predicate $\mathsf{spec}_{C_2\upharpoonright_R}$, K_1 the predicate $\mathsf{spec}_{C_1\upharpoonright_K}$ and finally K_2 the predicate $\mathsf{spec}_{C_2\upharpoonright_K}$. The proof of the fusion lemma requires establishing first a *mixed* determinism property on the loop's body predicates.

$$\forall\ s_1\ s_2\ s_1'\ s_2'.\ s_1 \equiv s_2 \wedge R_1(s_1, s_1') \wedge R_2(s_2, s_2') \implies s_1' \equiv s_2' \tag{4.6}$$

$$\forall\ s_1\ s_2\ s_1'\ s_2'.\ s_1 \equiv s_2 \wedge K_1(s_1, s_1') \wedge K_2(s_2, s_2') \implies s_1' \equiv s_2' \tag{4.7}$$

But, since by definition

$$\begin{aligned}
R_1(s, s') &= \exists s_0, s_1, s_2, s_3.\ s_0 \equiv s[\mathsf{tmp} \leftarrow [\![f_1 + f_2]\!](s)] \wedge \\
&\qquad s_1 \equiv s_0[f_1 \leftarrow [\![f_2]\!](s_0)] \wedge s_2 \equiv s_1[f_2 \leftarrow [\![\mathsf{tmp}]\!](s_1)] \ \wedge s_2 \equiv s' \\
&= s' \equiv s[f_1 \leftarrow [\![f_2]\!](s), f_2 \leftarrow [\![f_1 + f_2]\!](s)] \\
R_2(s, s') &= \exists s_0, s_1, s_2.\ s_0 \equiv s[f_2 \leftarrow [\![f_1 + f_2]\!](s)] \wedge \\
&\qquad s_1 \equiv s_0[f_1 \leftarrow [\![f_2 - f_1]\!](s_0)] \wedge s_1 \equiv s' \\
&= s' \equiv s[f_2 \leftarrow [\![f_1 + f_2]\!](s), f_1 \leftarrow [\![f_2]\!](s)] \\
K_1(s, s') &= K_2(s, s') = s' \equiv s[n \leftarrow [\![n - 1]\!](s)]
\end{aligned}$$

the proof that the properties 4.6 and 4.7 hold, is straightforward. We remark that the proof of such properties can be done automatically, using the Frama-c framework, simply by introducing the (mixed) lemma depicted in Figure 4.6 as a program annotation and using some automatic prover such as Alt-Ergo to discharge the related proof obligation. The proof of the lemma,

```
/*@ lemma body_equiv:
  @   \forall int n2_1, n2_2, n3_1, n3_2, f11_1, f11_2, f12_1, f12_2;
  @   \forall int tmp, f21_1, f21_2, f22_1, f22_2;
  @   n2_1 == n2_2 ==> f11_1 == f11_2 ==> f21_1 == f21_2 ==>
  @   loop_body_1(n2_1,n3_1,f11_1,f12_1,tmp,f21_1,f22_1) ==>
  @   loop_body_2(n2_2,n3_2,f11_2,f12_2,f21_2,f22_2) ==>
  @   (n3_1 == n3_2 && f12_1 == f12_2 && f22_1 == f22_2);
  @*/
```

Listing 4.6: Lemma which captures the property 4.6

$$\forall\ s_1\ s_2\ s'_1\ s'_2.\ s_1 \equiv s_2 \wedge \mathsf{loop}_1^{n_1}(s_1, s'_1)\ \wedge ([\![B]\!](s'_1) = \mathbf{false}) \wedge \mathsf{loop}_2^{n_2}(s_2, s'_2) \wedge ([\![B]\!](s'_2) = \mathbf{false}) \Longrightarrow s'_1 \equiv s'_2$$

is straightforward using the *mixed* synchronisation Lemma 16 and the property,

$$\forall\ s_1\ s_2\ s'_1\ s'_2.\ s_1 \equiv s_2 \wedge \mathsf{loop}_1^{n}(s_1, s'_1) \wedge \mathsf{loop}_2^{n}(s_2, s'_2) \Longrightarrow\ s'_1 \equiv s'_2$$

whose proof can be done by simple induction on n and using the properties 4.6 and 4.7 established above.

4.3 Verifying absence of error propagation

An important property of stream ciphers is their behavior when a bit in the ciphertext is flipped over the communication channel. The change may be due to a transmission error or maliciously introduced by an attacker. The way in which the decryption process reflects a wrong ciphertext symbol in the resulting plaintext is relevant: depending on the encryption scheme construction, a ciphertext error may simply lead to a corresponding flip in a plaintext symbol, or it may affect a significant number of subsequent symbols. This property, sometimes called *error propagation*, is usually taken as a criterion for selecting ciphers for noisy communication media, where the absence of error propagation can greatly increase throughput. For this purpose we studied the behavior of RC4 to evaluate if bit errors are not propagated in any way, i.e., if a ciphertext bit is flipped during the transmission, the only corresponding plaintext bit is affected. To this end, we formalised this property as an *information flow property*, using the *noninterference* definition (see Chapter 2).

Error propagation formalised as non-interference The intuition underlying the formalisation of error-propagation with non-interference is that secure information flow can be guaranteed by checking that arbitrary changes in low-integrity input variables cannot be detected by observing high-integrity output variables. We remark that the notion of a low-integrity input variable can be naturally associated with a transmission error over a communications channel. Hence, we map the i^{th} possibly erroneous ciphertext symbol to a non-trusted low-integrity input (we are looking at the decryption algorithm that, in the case of RC4, is identical to the one used for encryption). The definition of non-interference can then conveniently be used to capture the absence of error propagation. For this, we also associate the output plaintext symbols starting at position $i + 1$ to trusted high-integrity outputs. More precisely, our formulation captures the following idea: if an arbitrary change in the i^{th} input ciphertext symbol cannot be observed in the output plaintext symbols following position i, this implies that the stream cipher does not introduce error propagation in decryption.

Formally, considering V_H and V_L respectively as the sets of low-integrity input variables and high-integrity output variables of the RC4 implementation, and recalling the definition of noninterference from Chapter 2, we have for some $i \in [0, len[$:

$$V_H = \{\ \texttt{indata}[i]\ \}$$
$$V_L = \{\texttt{outdata}[j] \,|\, 0 \leq j < \ \texttt{len}\} \bigcup \texttt{key} \bigcup \{\texttt{indata}[j] \,|\, 0 \leq j < \texttt{len} \,\&\, j \neq i\,\}$$

We remark that the information-flow model of integrity can be treated as the dual to the confidentiality model, thus to apply the noninterference definition from Chapter 2, one has to consider the low-integrity inputs to be the set V_H and high-integrity outputs to be V_L. To verify if the RC4 implementation from openSSL enforces this property using Frama-c, we applied the self-composition approach (see Chapter 2). However, even for such a small example, it was a non-trivial exercise. The limitations that we encountered were essentially due to the growth of the problem size due to the aggressive optimisations that were used in the RC4 implementation in openSSL: (1) the use of macros rather than (inline) function calls leads to a source code size expansion; (2) the use of loop unrolling leads to intricate control flow inside the function (namely the extensive use of the `break` statement); and (3) the use of pointer arithmetic greatly increases the complexity of the generated proof obligations. For this reason, a refactoring of the code was required in order to achieve the goals that we set out in the beginning of this section.

```
unsigned char RC4NextKeySymbol(RC4_KEY *key) {
  unsigned char *d,x,y,tx,ty;

  x=key->x; y=key->y; d=key->data;
  x=((x+1)&0xff);      tx=d[x];
  y=(tx+y)&0xff;       d[x]=ty=d[y];
  d[y]=tx;  key->x=x; key->y=y;
  return d[(tx+ty)&0xff];
}

void RC4(RC4_KEY *key, const unsigned long len,
     const unsigned char *indata, unsigned char *outdata) {
  int i=0;
  while(i<len) { outdata[i]=indata[i] ^ RC4NextKeySymbol(key); i++; }
}
```

Listing 4.7: RC4 reference implementation

Listing 4.7 shows the version of RC4 we used.

The Frama-c input file used to show that the RC4 function above does not introduce error propagation is shown in Listings 4.8 and 4.9. The function is composed with itself and disjoint sets of variables are created for the two copies of the function, which is parametrised with the position i in which a transmission error could occur. The preconditions imposed on the composed function establish the equality of the high-integrity input variables for both copies of the function: all input variables except position i in the `indata` buffers. The postcondition on the composed function requires that the high-integrity output variables have equal values upon termination.

The verification of this code with Frama-c resulted in the generation of 18 proof obligations, all of which are automatically discharged by the back-end prover Simplify. This is made possible by the inclusion of the self-composition lemma, `spec1_sc` (proved offline with Coq following [7]) in the ACSL annotations.

4.4 Verifying correctness with respect to reference implementations

A direct transcription of the RC4 specification presented in the beginning of this chapter to a C implementation could look something like the code in Listing 4.7. Although this implementation is quite readable, and arguably verifiable by inspection, it was created without the slightest consideration for efficiency. This stands in contrast with the openSSL implementation of RC4 (see Appendix A.1) where readability (and

```
typedef struct rc4_key_st {unsigned char x, y; unsigned char data[256];}RC4_KEY;

/*@ predicate eqCondByK{L1,L2}(integer k, unsigned char *u1,unsigned char *u2) =
  @    \forall integer l; l!=k ==> \at(u1[l],L1)==\at(u2[l],L2);
  @*/

/*@ predicate eqArrays{L1,L2}(unsigned char *u1,unsigned char *u2) =
  @   \forall integer l; \at(u1[l],L1)==\at(u2[l],L2);
  @*/

/*@ axiomatic RC4KeyAxiom {
  @   logic unsigned char RC4KeyLogic{L1,L2}(unsigned char *x, unsigned char *y,
  @                                 unsigned char *d);
  @ axiom RC4KeyLogic_unique{L1,L2,L3,L4}:
  @  \forall unsigned char *x, unsigned char *y,unsigned char *d,
  @  unsigned char *x1, unsigned char *y1, unsigned char *d1;
  @  \at(*x,L1)==\at(*x1,L3) ==> \at(*y,L1)==\at(*y1,L3)  ==>
  @  eqArrays{L1,L3}(d,d1) ==>
  @  (RC4KeyLogic{L1,L2}(x,y,d) == RC4KeyLogic{L3,L4}(x1,y1,d1) &&
  @  \at(*x,L2)==\at(*x1,L4) && \at(*y,L2)==\at(*y1,L4) &&
  @  eqArrays{L2,L4}(d,d1));
  @ }
  @*/

/*@ inductive spec1{L1,L2}(integer i1,integer i2,integer k, unsigned char *inp,
  @   unsigned char *out,unsigned char *x, unsigned char *y,unsigned char *d) {
  @ case spec1_base{L}:\forall integer i1,integer i2,integer k,
  @   unsigned char *inp, unsigned char *out, unsigned char *x,
  @   unsigned char *y, unsigned char *d;
  @   i1 == i2 ==> spec1{L,L}(i1,i2,k,inp,out,x,y,d);
  @
  @ case spec1_nat{L1,L2,L3} :
  @   \forall integer i1,integer i2,integer i3,integer k, unsigned char *inp,
  @   unsigned char *out, unsigned char *x, unsigned char *y, unsigned char *d;
  @   spec1{L1,L2}(i1,i2,k,inp,out,x,y,d) ==> i3 == i2 +1 ==>
  @   eqCondByK{L2,L3}(k,inp,inp) ==>
  @   \at(out[i2],L3) == (\at(inp[i2],L2) ^ RC4KeyLogic{L2,L3}(x,y,d)) ==>
  @   spec1{L1,L3}(i1,i3,k,inp,out,x,y,d);
  @ }
  @*/

/*@ lemma spec1_sc{L1,L2,L3,L4} :
  @  \forall integer i,integer i1, integer i2, unsigned char *inp1,
  @   unsigned char *inp2, unsigned char *out1,unsigned char *out2,
  @   unsigned char *x1, unsigned char *x2, unsigned char *y1, unsigned char *y2,
  @   unsigned char *d1, unsigned char *d2, integer k;
  @   spec1{L1,L2}(i,i1,k,inp1,out1,x1,y1,d1) ==>
  @   spec1{L3,L4}(i,i2,k,inp2,out2,x2,y2,d2) ==>
  @   eqCondByK{L1,L3}(k,inp1,inp2) ==> eqArrays{L1,L3}(out1,out2) ==>
  @   eqArrays{L1,L3}(d1,d2) ==> \at(*x1,L1)==\at(*x2,L3) ==>
  @   \at(*y1,L1)== \at(*y2,L3) ==> i1 == i2 ==>
  @   (eqCondByK{L2,L4}(k,inp1,inp2) && eqArrays{L2,L4}(d1,d2) &&
  @   \at(*x1,L2)==\at(*x2,L4) && \at(*y1,L2)== \at(*y2,L4) &&
  @   eqCondByK{L2,L4}(k,out1,out2));
  @*/

//@ ensures \result == RC4KeyLogic{Old,Here}(&(key->x),&(key->y),key->data);
unsigned char RC4NextKeySymbol(RC4_KEY *key) {
  register unsigned char *d; register unsigned char x, y, tx, ty;
  x = key->x; y = key->y; d = key->data; x = ((x + 1) & 0xff); tx = d[x];
  y = (tx + y) & 0xff; d[x] = ty = d[y]; d[y] = tx; key->x = x; key->y = y;
  return d[(tx + ty) & 0xff]; }
```

Listing 4.8: Annotations and auxiliary functions to verify absence of error propagation in the RC4 implementation

```
/*@ requires len>=0 && len1>=0 && len==len1 &&
  @  eqCondByK{Here,Here}(k,indata,indata1) &&
  @  eqArrays{Here,Here}(key->data,key1->data) &&
  @  key->x==key1->x && key->y==key1->y &&
  @  eqArrays{Here,Here}(outdata,outdata1);
  @ ensures len==len1 && eqCondByK{Here,Here}(k,indata,indata1) &&
  @    eqArrays{Here,Here}(key->data,key1->data) &&
  @    key->x==key1->x && key->y==key1->y &&
  @    eqCondByK{Here,Here}(k,outdata,outdata1);
  @*/
void RC4_SC(RC4_KEY *key, const unsigned long len, const unsigned char *indata,
    unsigned char *outdata, RC4_KEY *key1, const unsigned long len1,
    const unsigned char *indata1, unsigned char *outdata1, int k) {
    int i = 0; int i1 = 0;

    /*@ loop invariant 0<=i<=len &&
      @     spec1{Pre,Here}(0,i,k,indata,outdata,&(key->x),&(key->y),key->data);
      @ loop variant (len-i);
      @*/
   while (i < len) { outdata[i] = indata[i] ^ RC4NextKeySymbol(key); i++; }

   /*@ loop invariant 0<=i1<=len1 &&
     @   spec1{Pre,Here}(0,i1,k,indata1,outdata1,&(key1->x),&(key1->y),key1->data);
     @ loop variant (len1-i1);
     @*/
   while (i1 < len1) { outdata1[i1] = indata1[i1] ^ RC4NextKeySymbol(key1);i1++;}
}
```

Listing 4.9: Annotated RC4 implementation to verify absence of error propagation

the inherent assurance of correctness) was sacrificed to achieve better performance.

This example supports the domain-specific motivation for the discussion presented in this thesis: the natural way to obtain assurance that an implementation of a cryptographic algorithm is correct, is to verify that it is functionally equivalent to another (more readable) implementation of the same algorithm. We have investigated how this goal can be achieved for the particular case of RC4, by identifying refactoring steps that may require a proof of equivalence in order to establish the correctness of different RC4 implementations.

We present next our approach to verifying the identified class of equivalence relations using the Frama-c tool. The results obtained are, of course, not only applicable to implementations of other cryptographic algorithms, but also to other application domains where similar program transformations may be employed (as it is shown in Section 4.2.4 with the Fibonacci implementations).

A simple refactoring to capture key pre-processing. A possible refactoring of the RC4 specification in Listing 4.7 is suggested by a common optimisation performed when using stream ciphers. Indeed, one of the ways of speeding up the throughput of

```
void RC4(RC4_KEY *key, const unsigned long len,
              const unsigned char *indata, unsigned char *outdata) {
  unsigned char keystream[len];

  int i=0;
  while(i<len) { keystream[i] = RC4NextKeySymbol(key); i++; }

  i=0;
  while(i<len) { outdata[i]=indata[i] ^ keystream[i]; i++; }
}
```

Listing 4.10: RC4 implementation with key pre-processing

stream cipher processing is to compute (a portion of) the key stream before the plaintext is available (or the ciphertext if one is decrypting). This means that the encryption operation to be performed on-the-fly is then reduced to simple masking using an XOR operation, which can be done extremely fast. For sychronous ciphers such as RC4, the number of key stream bits that can be pre-computed can be arbitrarily large, as this is totally independent of the encrypted data. The version of RC4 presented in Listing 4.10 moves in this direction by separating the key stream generation process from the plaintext masking (or ciphertext unmasking process).

To prove the equivalence between the programs in Listings 4.7 and 4.10, one can apply the *equivalence by composition* technique. In short, considering the programs C_1 and C_2 respectively as the programs of Listings 4.10 and 4.7, the steps to apply this technique are the following:

1. create a program that results from the composition of C_1 with C_2, where all the variables of C_2 are renamed;

2. annotate the composed program with pre- and postconditions to express that all the input and output variables in both programs (C_1 and C_2) are equal;

3. annotate each loop with a loop invariant using the *natural invariants* technique (introduced in Section 4.2.1);

4. define a fusion lemma which expresses that the specifications of the loop invariants are equivalent. This is a mixed fusion lemma because the structure of C_1 separates the key stream generation process from the plaintext masking, thus having two loop invariants, whose conjunction must be equivalent to the loop invariant defined to C_2;

5. finally, before annotating the function RC4NextKeySymbol with a post-condition, we must define a logic function and axiomatise its properties, stating that for the same inputs it produces the same outputs. The postcondition predicate must express that the result of RC4NextKeySymbol is equal the result of the logic function. Establishing a postcondition such as this, we are expressing that the result of RC4NextKeySymbol enjoys the same properties as the result of the logic function. We remark that this postcondition can be automatically proved using an automatic theorem prover such as Yices SMT Solver[4].

Part of the annotated source code used to achieve this result is shown in Listings 4.11 and 4.12.

A sequence of refactorings leading to the `openssl` implementation. We now discuss a more elaborate sequence of refactoring steps that permit reaching the openSSL implementation of RC4 in Appendix A.1, departing from the reference implementation in Listing 4.7. The first refactoring step, leading to the RC4 function in Listing 4.13, is not very interesting from a verification point of view. It consists of a number of simple transformations:

1. removing the auxiliary function by inlining the corresponding code in the main function body;
2. rearranging local variables to match those in the openSSL implementation;
3. applying the transitivity property of assignments in C to combine two statements;
4. replacing modular operations by their equivalent bit-wise operations. A macro is also introduced to improve readability.

The next refactoring steps, leading to the version shown in Listing 4.14, are more interesting examples of transformations involving loop refactorings. Concretely, the main loop is first separated into two loops with the same body, which are sequentially composed to realise the original number of iterations. The first loop is then modified by explicitly composing the original body with itself 8 times, and altering the increments

[4] `http://yices.csl.sri.com/`

```
/*@ inductive spec1{L1,L2}(integer i1,integer i2, unsigned char *stream,
 @   unsigned char *x,unsigned char *y, unsigned char *d) {
 @ case spec1base{L} :
 @  \forall integer i1,integer i2,unsigned char *stream,
 @      unsigned char *x, unsigned char *y, unsigned char *d,integer len;
 @  i1 == i2 ==> spec1{L,L}(i1,i2,stream,x,y,d);
 @
 @ case spec1_nat{L1,L2,L3} :
 @  \forall integer i1,integer i2, integer i3,unsigned char *stream,
 @  unsigned char *x, unsigned char *y, unsigned char *d;
 @      spec1{L1,L2}(i1,i2,stream,x,y,d) ==> i3 == i2 +1 ==>
 @      \at(stream[i2],L3) == RC4KeyLogic{L2,L3}(x,y,d) ==>
 @  spec1{L1,L3}(i1,i3,stream,x,y,d); }
 @*/

/*@ inductive spec2{L1,L2}(integer i1,integer i2, unsigned char *inp,
 @   unsigned char *out,unsigned char *key) {
 @ case spec2_base{L} :
 @  \forall integer i1,integer i2,unsigned char *inp,
 @  unsigned char *out,unsigned char *key;
 @  i1 == i2 ==> spec2{L,L}(i1,i2,inp,out,key);
 @
 @ case spec2_nat{L1,L2,L3} :
 @  \forall integer i1,integer i2,integer i3, unsigned char *inp,
 @  unsigned char *out,  unsigned char *key;
 @  spec2{L1,L2}(i1,i2,inp,out,key) ==> i3 == i2 +1 ==>
 @  eqArrays{L2,L3}(inp,inp) ==> eqArrays{L2,L3}(key,key) ==>
 @  \at(out[i2],L3) == (\at(inp[i2],L3) ^ \at(key[i2],L3)) ==>
 @  spec2{L1,L3}(i1,i3,inp,out,key);
 @ }
 @*/

/*@ inductive spec3{L1,L2}(integer i1,integer i2,unsigned char *inp,
 @   unsigned char *out,unsigned char *x,unsigned char *y,unsigned char *d) {
 @ case spec3_base{L} :
 @  \forall integer i1,integer i2, unsigned char *inp,unsigned char *out,
 @  unsigned char *x, unsigned char *y, unsigned char *d;
 @  i1 == i2 ==> spec3{L,L}(i1,i2,inp,out,x,y,d);
 @
 @ case spec3_nat{L1,L2,L3} :
 @  \forall integer i1,integer i2,integer i3, unsigned char *inp,
 @  unsigned char *out, unsigned char *x, unsigned char *y, unsigned char *d;
 @  spec3{L1,L2}(i1,i2,inp,out,x,y,d) ==> i3 == i2 +1 ==>
 @  eqArrays{L2,L3}(inp,inp) ==>
 @  \at(out[i2],L3) == (\at(inp[i2],L2) ^ RC4KeyLogic{L2,L3}(x,y,d)) ==>
 @  spec3{L1,L3}(i1,i3,inp,out,x,y,d);
 @ }
 @*/

/*@ lemma fusion_lemma {L1,L2,L3,L4,L5} :
 @ \forall integer i1, integer i2, integer i3, integer i4,integer i5,
 @  integer i6, unsigned char *inp1,unsigned char *inp2, unsigned char *out1,
 @  unsigned char *out2,unsigned char *x1, unsigned char *x2, unsigned char *y1,
 @  unsigned char *y2, unsigned char *d1,unsigned char *d2, unsigned char *key;
 @  spec1{L1,L2}(i1,i2,key,x1,y1,d1) ==> spec2{L2,L3}(i3,i4,inp1,out1,key) ==>
 @  spec3{L4,L5}(i5,i6,inp2,out2,x2,y2,d2) ==>
 @ (i1==i3 && i3==i5 && i2==i4 && i4==i6) ==>
 @ eqArrays{L1,L4}(inp1,inp2) ==> eqArrays{L1,L4}(d1,d2) ==>
 @ \at(*x1,L1) == \at(*x2,L4) ==> \at(*y1,L1) == \at(*y2,L4) ==>
 @ eqArrays{L3,L5}(out1,out2);
 @*/
```

Listing 4.11: RC4 refactoring for key pre-processing – natural invariants specification and fusion lemma

```
/*@ requires len==len1 && eqArrays{Here,Here}(indata,indata1) &&
 @   eqArrays{Here,Here}((unsigned char*)key->data,
 @   (unsigned char*)key1->data) && key->x==key1->x &&
 @   key->y==key1->y;
 @   ensures eqArrays{Here,Here}(outdata,outdata1);
 @*/
void RC4_SC(RC4_KEY *key, const unsigned long len,
  const unsigned char *indata,
  unsigned char *outdata, RC4_KEY *key1,
  const unsigned long len1,
  const unsigned char *indata1,
  unsigned char *outdata1,
  unsigned char *keystream)
{

  int i, i1; i1 = 0;

  /**************************Pre-processing************************/

  i = 0;

 /*@ loop invariant 0<=i<=len && spec1{Pre,Here}(0,i,keystream,
   @ (unsigned char *)&(key->x), (unsigned char *)&(key->y),
   @ (unsigned char *)key->data);
   @ loop variant (len-i);
   @*/
   while (i < len) {
     keystream[i] = RC4NextKeySymbol(key);
     i++;
   }

   //@ ghost goto L;
   //@ ghost L:
   i = 0;

   /*@ loop invariant 0<=i<=len && spec2{L,Here}(0,i,indata,outdata,keystream);
     @ loop variant (len-i);
     @*/
   while (i < len) {
     outdata[i] = indata[i] ^ keystream[i];
     i++;
   }

   /****************************Specification**************************/

   /*@ loop invariant 0<=i1<=len1 && spec3{Pre,Here}(0,i1,indata1,outdata1,
     @ (unsigned char *)&(key1->x),(unsigned char *)&(key1->y),
     @ (unsigned char *)key1->data);
     @ loop variant (len1-i1);
     @*/
   while (i1 < len1) {
     outdata1[i1] = indata1[i1] ^ RC4NextKeySymbol(key1);
     i1++;
   }
}
```

Listing 4.12: RC4 refactoring for key pre-processing

```
void RC4(RC4_KEY *key, const unsigned long len,
     const unsigned char *indata, unsigned char *outdata) {
  unsigned char x,y,tx,ty, *d;
  int i;
  x = key->x; y = key->y; d = key-> data;

  i=0;
  while(i<len) { RC4LOOP(indata,outdata,i); i++; }
  key->x=x; key->y=y;
}
```

Listing 4.13: RC4 refactoring step 1

```
void RC4(RC4_KEY *key, const unsigned long len,
         const unsigned char *indata, unsigned char *outdata) {
  unsigned char x,y,tx,ty, *d; int i;
  x = key->x; y = key->y; d = key-> data;
  i= (int)(len>>3L);
  while(i>0) {
    RC4LOOP(indata,outdata,0);
    RC4LOOP(indata,outdata,1);
    RC4LOOP(indata,outdata,2);
    RC4LOOP(indata,outdata,3);
    RC4LOOP(indata,outdata,4);
    RC4LOOP(indata,outdata,5);
    RC4LOOP(indata,outdata,6);
    RC4LOOP(indata,outdata,7);
    indata+=8; outdata+=8; i--;
  }

  i=(int)(len&0x07);
  while(i>0) {RC4LOOP(indata,outdata,i); i--; }
  key->x=x; key->y=y;
}
```

Listing 4.14: RC4 refactoring step 2

accordingly. The annotated code that was used to construct a proof of equivalence corresponding to the jump between these more elaborate refactoring steps can be found in `http://crypto.di.uminho.pt/CACE`. It is itself divided into two intermediate steps, in order to tame the complexity of the exercise.

The final refactoring steps, leading to the openSSL version of RC4 in Appendix A.1, are introduced to achieve additional speed-ups. Firstly, pointer arithmetic is used to reduce the range of indexing operations, and loop counting is inverted. Then, different control flow constructions are applied: all `while` loops are reformulated using the `break` statement to remove the final backward jump, and `if` constructions are introduced to detect termination cases.

```
static void mulmod(unsigned int h[17], const unsigned int r[17]) {
   unsigned int hr[17]; unsigned int i; unsigned int j; unsigned int u = 0;
   for (i = 0; i < 17; ++i) {
     u = 0;
     for (j = 0; j <= i; ++j)  u += h[j] * r[i   j];
     for (j = i + 1; j < 17; ++j)  u += 320 * h[j] * r[i + 17   j];
     hr[i] = u;
   }
   for (i = 0; i < 17; ++i) h[i] = hr[i];
   squeeze(h);
}
```

Listing 4.15: Example extracted from the crypto-core NaCl library

4.5 Defining natural invariants for loops in general

In this section we show how the natural invariants technique can be extended to more complex examples, such as loop-bodies which also contain loops. For this, we consider the example depicted in Figure 4.15, extracted from the core of the NaCl library, which contains a loop whose loop-body contains two inner loops. The intuition underlying the definition of this natural invariant for the outermost loop is given next.

The first step is to define a natural invariant for each innermost loop, as shown in Figure 4.16, where loop_2 and loop_3 correspond to the natural invariants for the first and second innermost loops, respectively. Recall that the definition of the natural invariant predicate has a base case and an inductive case. The base case is straightforward. To define the inductive case, we need first to identify the intermediate states introduced in its loop body. The easiest way is to introduce a new state after each instruction. As we can see below, in this case three intermediate states are introduced (where $loop_1$ corresponds to the outermost loop and $loop_2$ and $loop_3$ correspond to the first and second innermost loops, respectively).

$$\begin{tikzcd}
 & s'_1 \arrow[r, "loop_2"] & s'_2 \arrow[r, "loop_3"] & s'_3 \arrow[dr, bend left, "{hr[i] = u}"] & \\
s_1 \arrow[ur, bend left, "u = 0"] \arrow[rrrr, "loop_1"'] & & & & s_2
\end{tikzcd}$$

The assignment $u = 0$ introduces the first intermediate state S'_1. The execution of the first innermost loop ($loop_2$) introduces the second intermediate state S'_2 and the execution of the second innermost loop introduces the last intermediate state S'_3. All these intermediate states must be included in the inductive definition. However, new

```
/*@ inductive loop_2{L1,L2}(integer j1, integer j2, integer i,
 @     unsigned int u1, unsigned int u2, unsigned int *h, unsigned int *r){
 @   case base_case_loop_2{L}: \forall integer j1,j2,i;
 @       \forall unsigned int u1,u2,*h,*r;
 @     j1==j2 ==> u1==u2 ==> loop_2{L,L}(j1,j2,i,u1,u2,h,r);
 @   case ind_case_loop_2{L1,L2,L3}:
 @     \forall integer j1,j2,j3,i; \forall unsigned int u1,u2,u3,*h,*r;
 @        loop_2{L1,L2}(j1,j2,i,u1,u2,h,r) ==>
 @        j3 == j2 + 1 &&
 @     u3 == u2 + (\at(h[j2],L2)*\at(r[i  j2],L2));
 @     j2 <= i ==> loop_2{L1,L3}(j1,j3,i,u1,u3,h,r);
 @ }
 @*/

/*@ inductive loop_3{L1,L2}(integer j1, integer j2, integer i,
 @     unsigned int u1, unsigned int u2, unsigned int *h, unsigned int *r){
 @   case base_case_loop_3{L}: \forall integer j1,j2,i;
 @       \forall unsigned int u1,u2,*h,*r;
 @     j1==j2 ==> u1==u2 ==> loop_3{L,L}(j1,j2,i,u1,u2,h,r);
 @   case ind_case_loop_3{L1,L2,L3}:
 @     \forall integer j1,j2,j3,i; \forall unsigned int u1,u2,u3,*h,*r;
 @     loop_3{L1,L2}(j1,j2,i,u1,u2,h,r) ==>
 @        j3 == j2 + 1 && u3 == u2 + 320 * \at(h[j2],L2) * \at(r[i + 17   j2],L2);
 @     j2 < 17 ==> loop_3{L1,L3}(j1,j3,i,u1,u3,h,r);
 @ }
 @*/
```

Listing 4.16: Natural invariant for the innermost loops

labels are not always needed. In fact, if the variables modified in those states are not pointers or arrays, it suffices the introduction of a new fresh variable for each state. This is exactly what happens in this case, as can be seen in the ACSL definition of the natural invariant of the loop.

```
/*@ inductive loop_1{L1,L2}(integer i1, integer i2,
 @     unsigned int u, unsigned int *h, unsigned int *r, unsigned int *hr){
 @   case base_case_loop_1{L}:
 @     \forall integer i; \forall unsigned int u,*h,*r,*hr;
 @     loop_1{L,L}(i,i,u,h,r,hr);
 @   case ind_case_loop_1{L1,L2,L3,L4}:
 @     \forall integer i1,i2,i3,j,k; \forall unsigned int u1,u2,u3,u4,*h,*r,*hr;
 @     loop_1{L1,L2}(i1,i2,u1,h,r,hr) ==> u2 == 0 ==>
 @     loop_2{L2,L3}(0,j,i2,u2,u3,h,r) ==>
 @     loop_3{L3,L4}(i2+1,k,i2,u3,u4,h,r) ==>
 @     \at(hr[i2],L4) == u4 ==> i3 == i2 + 1 ==> i2 < 17 ==>
 @     loop_1{L1,L4}(i1,i3,u4,h,r,hr);
 @ }
 @*/
```

The only value that is modified between each intermediate state is the value of u. Since u is an integer variable, the variables u2, u3 and u4 represent its value in states S'_1,

S'_2 and S'_3, respectively. The label L3 is only introduced to refer to the state S'_2 in the inductive predicates of the innermost loops.

Annotating now the source code with the natural invariants defined for each loop, implies the introduction of labels which capture the intermediate states identified above. These labels can be introduced in the source code by means of ghost code annotations, using the ACSL notation.

```
    //@ ghost goto L1;
    //@ ghost L1:
    /*@ loop invariant loop_1{L1,Here}(0,i,u,h,r,hr);
      @ loop variant 17 i;
      @*/
    for (i = 0; i < 17; ++i) {
    u = 0;
    //@ ghost goto L2;
    //@ ghost L2:
    /*@ loop invariant loop_2{L2,Here}(0,j,i,\at(u,L2),u,h,r);
      @ loop variant i j;
      @*/
    for (j = 0; j <= i; ++j) { u += h[j] * r[i - j]; }

    //@ ghost goto L3;
    //@ ghost L3:
    /*@ loop invariant loop_3{L3,Here}(i+1,j,i,\at(u,L3),u,h,r);
      @ loop variant 17 j;
      @*/
    for (j = i + 1; j < 17; ++j) { u += 320 * h[j] * r[i + 17   j];}
    hr[i] = u;
}
```

The state needed to annotate each loop invariant with the corresponding natural invariant predicate is referenced by the label that immediately precedes it (label L1 represents the state S_1 and labels L2 and L3 represent the states S'_1 and S'_2, respectively).

The proof of composition-based lemmas can be easily done, if the definition of the natural invariants for such implementations follows the steps described above.

4.6 Summary

Summing up, we have focused on the verification of three security-relevant properties of real-world cryptographic implementations (using Frama-c), with increasing degrees of verification complexity: (1) safety properties (absence of numeric errors and memory safety); (2) absence of error propagation formalised as non-interference;

and (3) functional equivalence with respect to a reference implementation. Firstly we have proved that the RC4 implementation does not cause null pointer de-referencing exceptions, always performs array accesses with valid indices and computations do not overflow (Section 4.1). In other words, the implementation is secure against *buffer overflow* attacks. Then we have studied the behaviour of stream ciphers (such as RC4) when a bit in the ciphertext is flipped over a communication channel. The behaviour of RC4 is common to other *synchronous* ciphers: bit errors are not propagated in any way, i.e. if a ciphertext bit is flipped during transmission, then only the corresponding plaintext bit is affected. We have formalised this property as a novel application of the *non-interference* concept, and subsequently proved that the RC4 implementation indeed enjoys this property (Section 4.3). Finally, we have generalised the self-composition method [17] to prove the correctness of real implementations with respect to reference implementations (Section 2.3.3). Cryptography is a prime candidate for equivalence proofs, since specifications are usually given as reference implementations rather then using some high level model or language. In concrete terms we have proved the equivalence between a reference implementation of RC4 and the realistic implementation included in `openSSL` (Section 4.4).

Chapter 5

Verifying side-channel countermeasures

In this chapter, we extend the range of applications of the methods introduced in our previous chapter, to cope with a set of high-level non-functional security policies adopted by the developers of the NaCl cryptographic library. These policies, enforce software countermeasures against (timing) side-channel attacks.

5.1 Side-channel attacks

One of the most challenging aspects of cryptographic software implementation is the fact that functional correctness is *not* a sufficient condition to guarantee security. It is possible (and likely) that a naive implementation of a theoretically secure cryptographic algorithm is functionally correct, and yet turns out to be insecure. This is because cryptographic algorithms are designed and validated, in theory, by idealizing the computational platform in which they will execute: computation is seen as taking place inside a black box, from which only explicitly released outputs can be extracted. In practice, this is far from the truth, as physical observation of computational platforms can enable an adversary to recover sensitive information, often with very little effort. This type of attack is usually called a *side-channel attack*.

Protection against side-channel attacks is one of the most active areas of research in applied cryptography, involving both hardware and software implementation aspects. On the hardware side, the goal is to devise a platform that aproximates the idealized black-box mentioned above. Smart-cards, for example, incorporate various hardware

No data-dependent branches. *The CPU's instruction pointer, branch predictor, etc. are not designed to keep information secret. For performance reasons this situation is unlikely to change. The literature has many examples of successful timing attacks that extracted secret keys from these parts of the CPU.* NaCl *systematically avoids all data flow from secret information to the instruction pointer and the branch predictor. There are no conditional branches with conditions based on secret information; in particular, all loop counts are predictable in advance. This protection appears to be compatible with extremely high speed, so there is no reason to consider weaker protections.*

No data-dependent array indices. *The CPU's cache, TLB, etc. are not designed to keep addresses secret. For performance reasons this situation is unlikely to change. The literature has several examples of successful cache-timing attacks that used secret information leaked through addresses.* NaCl *systematically avoids all data flow from secret information to the addresses used in load instructions and store instructions. There are no array lookups with indices based on secret information; the pattern of memory access is predictable in advance. The conventional wisdom for many years was that achieving acceptable software speed for AES required variable-index array lookups, exposing the AES key to side-channel attacks, specifically cache-timing attacks. However, the paper "Faster and timing-attack resistant AES-GCM" by Emilia Käsper and Peter Schwabe at CHES 2009 introduced a new implementation that set record-setting speeds for AES on the popular Core 2 CPU despite being immune to cache-timing attacks.* NaCl *reuses these results.*

Figure 5.1: NaCl Security Policies

countermeasures to reduce exposure to side-channel attacks, e.g. by minimizing power consumption fluctuations when different operations are executed by the processor. However, it is not realistic to assume that one can resort to special purpose hardware whenever one needs to employ cryptography. Furthermore, hardware countermeasures are, by design, meant to thwart specific forms of physical data collection, which means that there is always room for new sources of leakage to be uncovered and exploited. Finally, even the most advanced cryptographic hardware cannot protect against side channel leakage caused by bad software implementation choices.

5.2 A formal verification-based approach

Software side-channel countermeasures aim to minimize the correlation between the sensitive inputs to the algorithm and physically observable variations in the behavior of the underlying computational platform, when the algorithm is executing. In this work we focus on a particular class of countermeasures aiming to eliminate timing dependencies, in both execution and memory access times, that may give rise to so-called timing attacks. In particular, we address a set of non-functional security policies, quoted in

Figure 5.1, adopted by the developers of the NaCl cryptographic library. Concretely, code is written so as to ensure that the sequence of executed instructions (i.e. the control flow) and the sequence of accessed memory addresses are independent of the sensitive inputs. We refer the interested reader to [58] for details on programming techniques that make this possible without forsaking performance.

We consider deductive formal verification as a means to obtain further guarantees that these side-channel countermeasures are correctly deployed in cryptographic software implementations. In practice, these guarantees are important not only for the end-users of the code, but also for developers working, say, in collaborative projects, in which the eligibility of contributions must be analyzed with respect to well-defined code quality criteria. Transferral of an increased level of assurance to a third party may be necessary, for example, in the context of software certification processes.

Our strategy to formally verify compliance to security policies such as those described above, which enforce the elimination of control flow and memory access dependencies as countermeasures against timing side-channel attacks, is to view them as information flow security restrictions. Our approach can be summarized as follows:

1. We extend the operational semantics of the While^C language (introduced in Chapter 2), to explicitly capture the flavour of side-channel leakage addressed by the NaCl security policies (quoted in Figure 5.1 from the NaCl specifications). Concretely, the semantics constructs traces of the memory addresses read or written to by a program, including program and data memory. Based on this, we propose a definition of *secure program* in the sense intended by the NaCl developers. This is essentially a *termination-sensitive non-interference* requirement stating that the address traces should be independent of secret data. Technically, our security notion can be seen as an extension of the Program Counter Model of [68, 87], where we add the capability to handle a wider range of attacks, including cache timing attacks [77] and branch prediction analysis attacks [2] by extending the model to cover data memory access patterns.

2. To formally verify that a program meets the previous definition of security it then suffices to proceed with the following two steps:

   (a) One transforms the original program P into one that explicitly collects in its output state (minimal) additional information about the execution of P; and

(b) One then formally verifies (using the composition-based methods introduced in the previous chapter) that this extra information is independent of secret data.

We theoretically validate this technique by showing that a proof of safety (including termination) of a program, and a proof of non-interference for the corresponding transformed program, together imply that the original program is indeed secure with respect to the intended security policy. The details are described in Section 5.3.

3. Finally, we discuss how our proof techniques can be deployed using real-world deductive verification tools, namely the Frama-c framework. We cover in Section 5.4 practical examples extracted form the NaCl cryptographic library, highlighting the potential for automation of the program transformation and self-composition proofs using natural invariants. In doing so, we answer questions raised in [87, 88] regarding the feasibility of addressing these problems using off-the-shelf verification tools. Concretely, we show that it is possible to carry out verification directly over the composed program, for a much wider class of programs than was previously achieved. Furthermore, we do not need to transform the input program into a more convenient form that goes around the limitations of the verification framework. In Section 5.4 we further elaborate on the differences and improvements with respect to related work.

5.3 Formalisation and verification of side-channel countermeasures

We illustrate now how the framework presented in Chapter 4 can be used to attest adherence to non-functional security policies. We start by explaining in Section 5.3.1 how the security policies put forward by the developers of the NaCl library can be understood semantically as a non-interference-like property, that cannot be expressed using the standard semantics of *While*^{C} language. In Section 5.3.2 we then instrument the semantics of the language (adding memory and control-flow traces) and use it in Section 5.3.3 to faithfully capture the policies under scrutiny. Section 5.3.4 applies a simple program transformation to reify the instrumented semantics (by internalising trace information in the programs), which allows expressing security as a standard non-interference property.

5.3.1 Security policy as a semantic property

Consider the *While*^{C} language introduced in Chapter 2. Let C be a safe program in *While*^{C}, H a set of high-security variables and $V_L = Vars \setminus H$. Then informally, C complies with the NaCl side-channel security policies if

> *for any two states s_1, s_2 such that $s_1 \overset{V_L}{=} s_2$, if $(C, s_1) \Downarrow s_1'$ then for some state s_2' one has that $(C, s_2) \Downarrow s_2'$, with the same memory trace and control flow for both executions.*

i.e., the memory locations accessed and the execution paths followed are equal for both initial states. More specifically, in order to the program to be safe according to the NaCl policies, these (memory trace and control-flow) cannot depend on the values of the secret inputs. This clearly ensures that the control flow and array lookups do not depend on secret information, as prescribed. Naturally, a plain state-based semantics does not allow expressing this property formally (since no trace information is manipulated), which motivates the introduction of an extended instrumented semantics.

5.3.2 Instrumented semantics

We consider two additions to the *While*^{C} language. Firstly, all commands except sequential composition are now labelled. This is equivalent to labeling every atomic statement and every boolean condition. We further assume that all considered programs are *well-labelled*, meaning that all the labels in a program are distinct. Labels can then be thought of as abstractions of the instruction-pointer to the corresponding code. Secondly, a new syntactic class of list-expressions is considered (together with the corresponding variables and assignment statements). Such lists are useless for programming, but they are convenient to capture the NaCl policies under a standard non-interference formulation, so we include them in the language and treat them consistently with the other constructions. Furthermore, our implementation in Frama-c of such lists is natural and consistent with this formalisation (see Section 5.4).

We will refer to this extended language as *While*$^{C^+}$. We remark that we restrict our attention to safe *While*$^{C^+}$ programs, i.e., safety is checked separately. Its syntax is given as follows.

Integers	$\mathsf{Int} \ni e^i$	$::=$	$\mathbf{n} \mid \mathtt{x}^i \mid e^i + e^i \mid e^i - e^i \mid e^i * e^i \mid e^i \mathbin{/} e^i \mid \mathtt{a}[e^i]$
Booleans	$\mathsf{Bool} \ni e^b$	$::=$	$\mathbf{true} \mid \mathbf{false} \mid \mathtt{x}^b \mid e^b == e^b \mid e^b\ != e^b \mid e^i < e^i \mid e^i > e^i$
List expressions	$\mathsf{List} \ni le$	$::=$	$\mathtt{nil} \mid \mathtt{cons}(e, le)$
Commands	$\mathsf{Comm} \ni C$	$::=$	$[\mathbf{skip}]^l \mid [\mathtt{x}:=e]^l \mid \left[\mathtt{a}[e^i]:=e^i\right]^l \mid [\mathtt{xl}:=le]^l \mid$
			$\left[\mathbf{if}\ (e^b)\ \{C_1\}\ \mathbf{else}\ \{C_2\}\right]^l \mid \left[\mathbf{while}\ (e^b)\ C\right]^l \mid C_1\mathtt{;}\ \ C_2$

We will use the notation $\mathsf{stmt}^C(l)$ to refer to the statement annotated with label l in program C (recall that labels are assumed to be distinct). Moreover, we remark that by construction every program should use a non-empty set of labels. We denote the leftmost label used in a program C by $\mathsf{firstLabel}(C)$.

To capture the memory locations accessed during the execution of a program, the operational semantics is instrumented in order to keep track of the sequence of performed accesses – the *memory trace*, ranged by γ. Each element of the memory trace consists of a pair $(\mathsf{v}, \mathit{offset})$ where v is the variable identifier and *offset* is the index of the accessed memory location (0 for non-array variables). The control-flow is also made explicit by computing the sequence of labels executed during the computation — the *control-flow trace*, ranged by δ. We will then consider judgements of the form $(C, s) \Downarrow (s', \delta, \gamma)$ meaning that program C executed in state s terminates in state s', having followed the control-flow path δ and performed memory accesses γ. An auxiliary judgment is used for expressions: $(e, s) \Downarrow^e (\mathbf{n}, \gamma)$ means that expression e evaluated in state s returns the value $\mathbf{n}$, having performed accesses γ. When the traces in the final configuration are not important they will be omitted as in $(C, s) \Downarrow s'$. Figure 5.2 presents the big-step rules for both expressions and programs, where ϵ denotes the empty sequence, $\cdot$ denotes concatenation of sequences, and the singleton sequence is identified with its element (e.g. $l \cdot \delta$ denotes the addition of l in front of δ).

We now state a few useful lemmas. A first observation is that the control-flow trace constrains significantly the memory access trace of any given program. If an execution path is fixed, only indices for array accesses are allowed to vary. Let us denote by $\mathsf{projFst}(\gamma)$ the function that projects the first component of a memory trace γ, returning a list of variable identifiers.

Lemma 18. *Let C be a program, e an expression, and s_1, s_2 states.*

$$\frac{}{(\mathbf{n}, s) \Downarrow^{e} (\mathbf{n}, \epsilon)} \qquad \frac{}{(\mathtt{x}^i, s) \Downarrow^{e} (s(\mathtt{x}^i), (\mathtt{x}^i, 0))} \qquad \frac{(e, s) \Downarrow^{e} (v, \gamma)}{(\mathtt{a}[e], s) \Downarrow^{e} (s(\mathsf{a})(v), (\mathsf{a}, v) \cdot \gamma)}$$

$$\frac{}{(\mathbf{true}, s) \Downarrow^{e} (\mathbf{true}, \epsilon)} \qquad \frac{}{(\mathbf{false}, s) \Downarrow^{e} (\mathbf{false}, \epsilon)} \qquad \frac{}{(\mathtt{x}^b, s) \Downarrow^{e} (s(\mathtt{x}^b), (\mathtt{x}^b, 0))}$$

$$\frac{(e_1, s) \Downarrow^{e} (v_1, \gamma_1) \qquad (e_2, s) \Downarrow^{e} (v_2, \gamma_2)}{(e_1 \; op \; e_2, s) \Downarrow^{e} (v_1 \; [\![op]\!] \; v_2, \gamma_1 \cdot \gamma_2)} \quad op \in \{+, -, *, /, ==, !=, <, >\}$$

$$\frac{}{(\mathtt{nil}, s) \Downarrow^{e} (\mathtt{nil}, \epsilon)} \qquad \frac{(e, s) \Downarrow^{e} (v, \gamma_1) \qquad (le, s) \Downarrow^{e} (lv, \gamma_2)}{(\mathtt{cons}(e, le), s) \Downarrow^{e} (\mathtt{cons}(v, lv), \gamma_1 \cdot \gamma_2)}$$

$$\frac{}{([\mathbf{skip}]^l, s) \Downarrow (s, l, \epsilon)}$$

$$\frac{(e_1, s) \Downarrow^{e} (v_1, \gamma_1) \qquad\qquad (e_2, s) \Downarrow^{e} (v_2, \gamma_2)}{([\mathtt{a}[e_1]\mathtt{:=}e_2]^l, s) \Downarrow (s[\mathtt{a} \leftarrow \mathsf{upd}(s(\mathsf{a}), v_1, v_2)], l, (\mathsf{a}, v_1) \cdot \gamma_1 \cdot \gamma_2)}$$

$$\frac{(e, s) \Downarrow^{e} (v, \gamma)}{([\mathtt{x}\mathtt{:=}e]^l, s) \Downarrow (s[\mathtt{x} \leftarrow v], l, (\mathtt{x}, 0) \cdot \gamma)} \qquad \frac{(le, s) \Downarrow^{e} (lv, \gamma)}{([\mathtt{xl}\mathtt{:=}le]^l, s) \Downarrow (s[\mathtt{xl} \leftarrow lv], l, (\mathtt{xl}, 0) \cdot \gamma)}$$

$$\frac{(e, s) \Downarrow^{e} (v, \gamma) \qquad (C_1, s) \Downarrow (s_1, \delta_1, \gamma_1)}{([\mathbf{if}\ (e)\ \{C_1\}\ \mathbf{else}\ \{C_2\}]^l, s) \Downarrow (s_1, l \cdot \delta_1, \gamma \cdot \gamma_1)} \quad \mathit{if}\ v \neq 0$$

$$\frac{(e, s) \Downarrow^{e} (v, \gamma) \qquad (C_2, s) \Downarrow (s_2, \delta_2, \gamma_2)}{([\mathbf{if}\ (e)\ \{C_1\}\ \mathbf{else}\ \{C_2\}]^l, s) \Downarrow (s_2, l \cdot \delta_2, \gamma \cdot \gamma_2)} \quad \mathit{if}\ v = 0$$

$$\frac{(e, s) \Downarrow^{e} (v, \gamma) \qquad (C, s) \Downarrow (s_1, \delta_1, \gamma_1) \qquad ([\mathbf{while}\ (e)\ C]^l, s_1) \Downarrow (s_2, \delta_2, \gamma_2)}{([\mathbf{while}\ (e)\ C]^l, s) \Downarrow (s_2, l \cdot \delta_1 \cdot \delta_2, \gamma \cdot \gamma_1 \cdot \gamma_2)} \quad \mathit{if}\ v \neq 0$$

$$\frac{(e, s) \Downarrow^{e} (v, \gamma)}{([\mathbf{while}\ (e)\ C]^l, s) \Downarrow (s, l, \gamma)} \quad \mathit{if}\ v = 0$$

$$\frac{(C_1, s) \Downarrow (s_1, \delta_1, \gamma_1) \qquad (C_2, s_1) \Downarrow (s_2, \delta_2, \gamma_2)}{(C_1; C_2, s) \Downarrow (s_2, \delta_1 \cdot \delta_2, \gamma_1 \cdot \gamma_2)}$$

Figure 5.2: Evaluation semantics

1. *If* $(e, s_1) \Downarrow^{e} (v_1, \gamma_1)$ *and* $(e, s_2) \Downarrow^{e} (v_2, \gamma_2)$, *then* $\mathsf{projFst}(\gamma_1) = \mathsf{projFst}(\gamma_2)$.

2. *If* $(C, s_1) \Downarrow (s_1', \delta, \gamma_1)$, $(C, s_2) \Downarrow (s_2', \delta, \gamma_2)$, *then* $\mathsf{projFst}(\gamma_1) = \mathsf{projFst}(\gamma_2)$.

Proof. (1) By structural induction on e. The only case that does not follow directly by induction hypothesis is the *access* of an array element. But, since we are projecting the first components of the memory access traces, the possibly distinct array indexes accessed are irrelevant. (2) Observe that the assumption of distinct labels in C together with the premise that both executions share the control-flow trace δ force the shape of both derivations to be equal (in particular, branching conditions are evaluated to the same truth value). Then, a simple induction on the structure of C allows us to conclude the argument (again, the only case that does not follow immediately from induction hypothesis and (1) is array assignment, and again the first component is state independent). □

Another way of looking at the previous lemma is to state that the differences between two memory traces γ_1, γ_2 obtained through the same execution path concern only the sequences of indexes accessed in one or more arrays. Denoting by $\mathsf{projArr}^{\mathsf{a}}(\gamma)$ the function that returns the list of indexes accessed in an array a, we have:

Lemma 19. *Let C be a program such that $(C, s_1) \Downarrow (s_1', \delta, \gamma_1)$ and $(C, s_2) \Downarrow (s_2', \delta, \gamma_2)$. Then, $\gamma_1 = \gamma_2$ if and only if for all array variables a in C, $\mathsf{projArr}^{\mathsf{a}}(\gamma_1) = \mathsf{projArr}^{\mathsf{a}}(\gamma_2)$.*

Proof. The left-to-right implication is trivial. For the converse, observe that the common execution trace in both final configurations implies, by Lemma 18, that $\mathsf{projFst}(\gamma_1) = \mathsf{projFst}(\gamma_2)$ (in particular, γ_1 and γ_2 have the same length). Now, assume that $\gamma_1 \neq \gamma_2$ and let γ' be the greatest common prefix of γ_1 and γ_2. Since $\gamma_1 \neq \gamma_2$, the length of γ' is strictly smaller than that of γ_1 and γ_2. Consider that the first element where both sequences diverge is now added to this prefix, i.e. $\gamma_1' = \gamma' \cdot (\mathsf{a}, v_1)$ and $\gamma_2' = \gamma' \cdot (\mathsf{a}, v_2)$ (again, by Lemma 18 we know that the first components are equal). By construction, $v_1 \neq v_2$ which implies that $\mathsf{projArr}^{\mathsf{a}}(\gamma_1) \neq \mathsf{projArr}^{\mathsf{a}}(\gamma_2)$. □

Control-flow traces are also severely constrained: there are specific points where different executions may diverge, which correspond exactly to the boolean conditions tests performed by the program (*if* and *while* statements).

Lemma 20. *Let C be a program such that $(C, s_1) \Downarrow (s_1', \delta_1, \gamma_1)$ and $(C, s_2) \Downarrow (s_2', \delta, \gamma_2)$. Then, $\delta_1 = \delta_2$ if and only if $\mathsf{tests}^C(\delta_1) = \mathsf{tests}^C(\delta_2)$.*

Function $\mathsf{tests}^C(\cdot)$ extracts the outcomes of these tests from a given execution trace.

$$\mathsf{tests}^C(\epsilon) = \epsilon$$

$$\mathsf{tests}^C(l\cdot\delta) = \begin{cases} \mathsf{tests}^C(\delta) & \text{if } \mathsf{stmt}^C(l) \text{ is not an } \textit{if} \text{ nor a } \textit{while} \\ 1\cdot\mathsf{tests}^C(\delta) & \text{if } \mathsf{stmt}^C(l) = [\textbf{if}\ (e)\ \{C_1\}\ \textbf{else}\ \{C_2\}]^l, \\ & \delta = l'\cdot\delta'' \text{ and } l' = \mathsf{firstLabel}(C_1) \\ 0\cdot\mathsf{tests}^C(\delta) & \text{if } \mathsf{stmt}^C(l) = [\textbf{if}\ (e)\ \{C_1\}\ \textbf{else}\ \{C_2\}]^l, \\ & \delta = l'\cdot\delta'' \text{ and } l' = \mathsf{firstLabel}(C_2) \\ 1\cdot\mathsf{tests}^C(\delta) & \text{if } \mathsf{stmt}^C(l) = [\textbf{while}\ (e)\ C_1]^l, \\ & \delta = l'\cdot\delta' \text{ and } l' = \mathsf{firstLabel}(C_1) \\ 0\cdot\mathsf{tests}^C(\delta) & \text{if } \mathsf{stmt}^C(l) = [\textbf{while}\ (e)\ C_1]^l \\ & \text{and either } \delta = \epsilon, \text{ or } \delta = l'\cdot\delta' \text{ and } l' \neq \mathsf{firstLabel}(C_1) \end{cases}$$

Proof. The left-to-right implication is trivial. For the converse, assume $\delta_1 \neq \delta_2$ and let δ' be the greatest common prefix of both traces. We firstly observe that δ' is nonempty (its first element is necessarily $\mathsf{firstLabel}(C)$), and that the last label of δ' must be the label of an *if* or *while* statement (in any other case, the control flow is state-independent and thus leads to a common follow-up on both executions). Summarising, we have $\delta_1 = \delta'\cdot\delta'_1$, $\delta_2 = \delta'\cdot\delta'_2$, $\delta' = \delta''\cdot l'$, l' is a label of an *if* or *while* statement and the greatest common prefix of δ'_1 and δ'_2 is ϵ. Since $\delta_1 \neq \delta_2$, it cannot be the case that both δ'_1 and δ'_2 are empty. Without loss of generality, assume δ'_1 is nonempty with l'_1 as its first element. Since δ'_1 and δ'_2 have ϵ as its greatest common prefix, l'_1 cannot be the first element of δ'_2, and hence $\mathsf{tests}^C(l'\cdot\delta'_1) \neq \mathsf{tests}^C(l'\cdot\delta'_2)$. It follows then that $\mathsf{tests}^C(\delta_1) \neq \mathsf{tests}^C(\delta_2)$. □

5.3.3 Formal security definition

The NaCl side-channel security policies (Figure 5.1) can now be expressed as a non-interference-like property.

Definition 21. *Let C be a program, V_H high-security variables and $V_L = Vars \setminus V_H$. We say that C is NaCl-secure if*

$$(s_1 \overset{V_L}{=} s_2 \wedge (C, s_1) \Downarrow (s'_1, \delta_1, \gamma_1)) \implies$$
$$\textit{For some } s'_2,\ \delta_2,\ \textit{and } \gamma_2,\quad (C, s_2) \Downarrow (s'_2, \delta_2, \gamma_2) \wedge (\delta_1 = \delta_2 \wedge \gamma_1 = \gamma_2).$$

A weaker termination-insensitive variant is also considered, namely

$$s_1 \overset{V_L}{=} s_2 \;\wedge (C, s_1) \Downarrow (s_1', \delta_1, \gamma_1) \;\wedge\; (C, s_2) \Downarrow (s_2', \delta_2, \gamma_2) \implies (\delta_1 = \delta_2 \;\wedge\; \gamma_1 = \gamma_2).$$

Analogously, an expression e is said to be NaCl-secure if

$$s_1 \overset{V_L}{=} s_2 \;\wedge (e, s_1) \Downarrow^e (v_1, \gamma_1) \implies$$
$$\text{For some } v_2 \text{ and } \gamma_2, \quad (e, s_2) \Downarrow^e (v_2, \gamma_2) \;\wedge\; \gamma_1 = \gamma_2 \;\wedge\; v_1 = v_2.$$

The following proposition captures a convenient compositional property of our security notion.

Proposition 22 (Compositionality)**.** *Let C_1 and C_2 be NaCl-secure programs, and let e_1 and e_2 be NaCl-secure expressions. Then,*

1. *e_1 op e_2, and $a[e_1]$ are NaCl-secure expressions;*

2. *$C_1; C_2$, $[\textbf{\textit{while}}\ (e_1)\ C_1]^l$ and $[\textbf{\textit{if}}\ (e_1)\ \{C_1\}\ \textbf{\textit{else}}\ \{C_2\}]^l$ are NaCl-secure programs;*

Proof. By structural induction on expressions and programs. Perhaps the most difficult case is the array access, where for two distinct states we have to prove that $(a, v_1) \cdot \gamma = (a, v_2) \cdot \gamma$ (v_1 and v_2 are the values of e_1 in the different states and γ the memory positions accessed by e_1). But the proof follows directly by induction hypothesis since it is assumed that e_1 is secure, then $v_1 = v_2$. □

The above property has implications on both the scalability and modularity of our techniques. We rely on it to conduct the formal verification exercise in a gradual way, starting from leaf functions, and tackling each function independently. This allows us to tame the complexity of each verification step and combine the results to obtain a global security guarantee. Furthermore, the results one obtains for a verified component (such as the NaCl library) are established once-and-for-all, and can be reused as an intermediate result in subsequent verification exercises, e.g. verifying different client applications that may come to use the NaCl library.

5.3.4 Verification of security

Although Definition 21 nicely captures the NaCl side-channel security policies, it is not a convenient formalization for our verification purposes: we aim to apply self-composition, and so we require a specification that expresses security directly over the program state. To this end, we now introduce a program transformation that internalises

$$\langle \mathbf{n} \rangle = \langle \mathtt{nil} \rangle = \langle \mathtt{x} \rangle = \langle \mathtt{xl} \rangle = [\mathbf{skip}]^{l}$$
$$(l \text{ a fresh label})$$
$$\langle \mathtt{a}[e] \rangle = \langle e \rangle \,;\, [\mathtt{xl}^{a} \mathtt{:=} \mathtt{cons}(e, \mathtt{xl}^{a})]^{l}$$
$$(l \text{ a fresh label})$$
$$\langle e_1 \; op \; e_2 \rangle = \langle e_1 \rangle \,;\, \langle e_2 \rangle$$
$$\langle \mathtt{cons}(e, le) \rangle = \langle e \rangle \,;\, \langle le \rangle$$

$$\left\langle [\mathbf{skip}]^{l} \right\rangle = [\mathbf{skip}]^{l}$$
$$\left\langle [\mathtt{x} \mathtt{:=} e]^{l} \right\rangle = \langle e \rangle \,;\, [\mathtt{x} \mathtt{:=} e]^{l}$$
$$\left\langle [\mathtt{xl} \mathtt{:=} le]^{l} \right\rangle = \langle le \rangle \,;\, [\mathtt{xl} \mathtt{:=} le]^{l}$$
$$\left\langle [\mathtt{a}[e_1] \mathtt{:=} e_2]^{l} \right\rangle = \langle e_1 \rangle \,;\, \langle e_2 \rangle \,;\, [\mathtt{xl}^{a} \mathtt{:=} \mathtt{cons}(e_1, \mathtt{xl}^{a})]^{l'} \,;\, [\mathtt{a}[e_1] \mathtt{:=} e_2]^{l}$$
$$(l' \text{ a fresh label})$$
$$\left\langle [\mathbf{if}\ (e)\ \{C_1\}\ \mathbf{else}\ \{C_2\}]^{l} \right\rangle = \langle e \rangle \,;\, [\mathtt{control} \mathtt{:=} \mathtt{cons}((e \ \mathtt{!=}\ \mathbf{false}), \mathtt{control})]^{l'} \,;$$
$$[\mathbf{if}\ (e)\ \{\langle C_1 \rangle\}\ \mathbf{else}\ \{\langle C_2 \rangle\}]^{l}$$
$$(l' \text{ a fresh label})$$
$$\left\langle [\mathbf{while}\ (e)\ C]^{l} \right\rangle = \langle e \rangle \,;\, [\mathtt{control} \mathtt{:=} \mathtt{cons}((e \ \mathtt{!=}\ \mathbf{false}), \mathtt{control})]^{l'} \,;$$
$$\left[\mathbf{while}\ (e)\ \ \langle C \rangle \,;\, \langle e \rangle \,;\, [\mathtt{control} \mathtt{:=} \mathtt{cons}(e, \mathtt{control})]^{l'_1}\right]^{l}$$
$$(l', l'_1 \text{ fresh labels})$$
$$\langle C_1 ; C_2 \rangle = \langle C_1 \rangle \,;\, \langle C_2 \rangle$$

Figure 5.3: Transformation for internalising trace information

into the program state sufficient information from the instrumented semantics. The transformed programs explicitly manipulate control-flow and memory access trace information.

Figure 5.3 contains the definition of the transformation $\langle \cdot \rangle$ for both expressions and programs. The transformation makes use of fresh list variables `control` and $\mathtt{xl}^{a}$ (for each array variable `a`). Informally, given an expression e and a command C, $\langle e \rangle$ is a program that stores the indexes of arrays accessed during the evaluation of e (in the corresponding variables $\mathtt{xl}^{a}$), and $\langle C \rangle$ is similar to C but also keeps track of all conditional tests performed and of all array access indexes (in variables `control` and $\mathtt{xl}^{a}$). The following proposition relates in precise terms the final values of these variables of the transformed program, and the memory and execution traces of the original.

Proposition 23. *Let C be a program such that $(C, s) \Downarrow (s', \delta', \gamma')$. Consider moreover* *that $\overline{s}^0$ is the state that assigns to variable* `control` *and* $\mathtt{xl}^a$ *(for every array variable* a *in C) the empty sequence ϵ. Then,* $(\langle C\rangle, s \uplus \overline{s}^0) \Downarrow \overline{s}$*, where:*

- $\overline{s} = s' \uplus \overline{s}'$*, with* $\mathsf{dom}(\overline{s}^0) = \mathsf{dom}(\overline{s}')$*,*
- $\overline{s}'(\mathtt{control}) = \mathsf{tests}^C(\delta')$*,*
- $\overline{s}'(\mathtt{xl}^a) = \mathsf{projArr}^{\mathsf{a}}(\gamma')$*.*

Proof. By structural induction on the derivation of $(C, s) \Downarrow (s', \delta', \gamma')$. It is clear from the definition of the transformation that the inserted code only affects variables introduced by it, hence the partition of the final state is immediate. Moreover, every conditional test performed during the execution is explicitly stored in variable `control` (notice that, for the case of *while* loops, the transformation inserts code before the loop and at the end of the loop body). Finally, every evaluated expression of the original program is preceded by the execution of the transformation of that same expression. □

Theorem 24. *Let C be a program, V_H high-security variables, $\langle V\rangle$ the set of variables introduced by transforming C to $\langle C\rangle$, and $\langle V_L\rangle = Vars(C) \setminus V_H \cup \langle V\rangle$. The program C is (termination-insensitive) secure with respect to Definition 21 if for states s_1 and s_2,*

$$(s_1 \overset{\langle V_L\rangle}{=} s_2 \;\wedge\; (\langle C\rangle, s_1) \Downarrow s_1' \;\wedge\; (\langle C\rangle, s_2) \Downarrow s_2') \implies s_1' \overset{\langle V\rangle}{=} s_2'$$

Proof. Follows directly from Proposition 23 and Lemmas 19 and 20.
By Proposition 23, $s_1 = s_1^i \uplus \overline{s}_1^0$ and $s_2 = s_2^i \uplus \overline{s}_2^0$ where $(C, s_1^i) \Downarrow s_1^f$ and $(C, s_2^i) \Downarrow s_2^f$. So, $s_1' = s_1^f \uplus \overline{s}_1'$ and $s_2' = s_2^f \uplus \overline{s}_2'$, where

- $s_1' = s_1^f \uplus \overline{s}_1'$, with $\mathsf{dom}(\overline{s}_1^0) = \mathsf{dom}(\overline{s}_1')$ and $s_2' = s_2^f \uplus \overline{s}_2'$, with $\mathsf{dom}(\overline{s}_2^0) = \mathsf{dom}(\overline{s}_2')$,
- $\overline{s}_1'(\mathtt{control}) = \mathsf{tests}^C(\delta_1)$ and $\overline{s}_2'(\mathtt{control}) = \mathsf{tests}^C(\delta_2)$,
- $\overline{s}_1'(\mathtt{xl}^a) = \mathsf{projArr}^{\mathsf{a}}(\gamma_1)$ and $\overline{s}_2'(\mathtt{xl}^a) = \mathsf{projArr}^{\mathsf{a}}(\gamma_2)$.

for all array a in C. By Definition 21, C is secure only if $\delta_1 = \delta_2$ and $\gamma_1 = \gamma_2$. By Lemma 20 if $\mathsf{tests}^C(\delta_1) = \mathsf{tests}^C(\delta_2)$ then $\delta_1 = \delta_2$, and if so, by Lemma 19 if $\mathsf{projArr}^{\mathsf{a}}(\gamma_1) = \mathsf{projArr}^{\mathsf{a}}(\gamma_2)$ then $\gamma_1 = \gamma_2$, for all a in C. Since $\overline{s}_1'(\mathtt{control}) = \mathsf{tests}^C(\delta_1)$ and $\overline{s}_2'(\mathtt{control}) = \mathsf{tests}^C(\delta_2)$ and $\overline{s}_1'(\mathtt{xl}^a) = \mathsf{projArr}^{\mathsf{a}}(\gamma_1)$ and $\overline{s}_2'(\mathtt{xl}^a) = \mathsf{projArr}^{\mathsf{a}}(\gamma_2)$, then to prove that the program is secure it is sufficient to prove that $\overline{s}_1'(\mathtt{control}) = \overline{s}_2'(\mathtt{control})$ and that $\overline{s}_1'(\mathtt{xl}^a) = \overline{s}_2'(\mathtt{xl}^a)$, for all array a in C, i.e., prove that $s_1 \overset{\langle V_L\rangle}{=} s_2$. So, the proof of this theorem implies that the program is secure according to Definition 21. □

The formulation given by Theorem 24 can be readily verified by the self-composition technique (see Chapter 4). A similar result could be derived for the termination-sensitive variant of security, but that would not be directly usable with self-composition. In our approach we separately handle the proof of termination, which together with the previous result trivially yields the termination-sensitive variant.

5.4 Case study: NaCl cryptographic library

We now present examples of how the techniques presented in the previous sections can be used in practice to formally verify compliance to these policies, using off-the-shelf verification tools. We selected two additional examples from the core of the NaCl library, aiming to highlight various aspects of our contributions. We begin with a simpler one, which we can describe in more detail to adequately illustrate the practical implementations aspects of our work. We then move on to discuss a more complex example to further justify our contributions. Overall, we have successfully applied these techniques to the formal verification of all of the core functions in the NaCl library (aprox. 560 loc). Nevertheless, and even though we argue that most of the annotation work required to carry out the exercise can be automated, we have manually annotated the programs. The discharge of the resulting verification conditions, with the exception of the loop-related lemmata that we explicitly factor out in the self-composition proofs, was fully handled by automatic provers.

5.4.1 A simple example

The selected function[1] is called crypto_verify and is presented in Listing 5.1. It may be surprising to know that the high-level specification for this function is that it compares the contents of two 16-byte arrays x and y, whose contents are high-security and must not be leaked. The introduced optimizations aim to ensure both control flow and data memory access independence, as prescribed by the NaCl security policies. As a side note, we remark that we have also verified that this function is functionally correct with respect to a (readable) reference implementation, shown in Listing 5.2, using the methodology introduced in the previous chapter. Appendix B contains the Frama-c annotated code to prove the equivalence between the implementations of Listing 5.1

[1]The actual implementation in the NaCl library totally unfolds the while loop, but this would not be as convenient for ilustrative purposes.

```
int crypto_verify(const unsigned char *x, const unsigned char *y)
{
   int differentbits = 0, i = 0;

   while (i < 16) { differentbits |= x[i] ^ y[i]; i++; }
   return (1 & ((differentbits - 1) >> 8)) - 1;
}
```

Listing 5.1: NaCl implementation of crypto_verify function

```
int crypto_verify(const unsigned char *x, const unsigned char *y) {
    int i = 0; int res = 0;
    while (i < 16) {  if (x[i] != y[i]) { res = -1;}  i++; }
    return res;
}
```

Listing 5.2: Reference implementation of crypto_verify function

and Listing 5.2 (the proof only required two refactoring steps). As can be seen, the more readable implementation of the crypto_verify function is not safe accordingly to our definition of security (Definition 21), and obviously susceptible to side-channel attacks. The control-flow of the program is influenced by the input values of x and y. Any attacker capable of observing the positions of the instruction pointer during the program execution, may find out the values stored in x and y.

As explained at the end of the previous section, we establish (termination-insensitive) security by splitting our formal verification exercise in two independent steps. The first step is to verify safety (and termination) for all valid inputs. The second step is to apply the program reification and formal verification tasks that permit applying Theorem 24 and establishing that the program is indeed secure according to Definition 21.

Safety and termination verification This step can be easily achieved in Frama-c by annotating the code with appropriate pre-conditions, imposing the validity of input arrays in the proper range, and adding some simple lemmas that allow the tool to recognize the correct output range of the bit-wise operations used. As in the proof of safety of the RC4 algorithm in the previous chatper, these lemmas are required because a sufficiently expressive axiomatic semantics for these operations is typically not included in off-the-shelf formal verification tools such as Frama-c, since such operations are rarely used in general-purpose software. Listing 5.3 depicts the Frama-c annotated code to prove the safety of crypto_verify. A variant to prove termination and an invariant to

```
/*@ lemma bw_xor: \forall unsigned char a, b; 0 <= (a ^ b) < 256;
  @
  @ lemma bw_or: \forall int  a, b; 0<=a<256 &&  0<=b<256 ==> 0 <=(a | b)<256;
  @
  @ lemma bw_ext: \forall int a; 0 <= a < 256 ==> 0 <= (1 & ((a-1)>>8)) <= 1;
  @*/

/*@ requires \valid(x+(0..15)) && \valid(y+(0..15));
 @*/
int crypto_verify_loop(const unsigned char *x, const unsigned char *y) {
     int differentbits = 0;  int i=0;

     /*@ loop invariant 0<= differentbits <256;
       @ loop variant 16-i;
       @*/
     while(i<16) { differentbits |= x[i] ^ y[i]; i++; }

     return (1 & ((differentbits - 1) >> 8)) - 1;
}
```

Listing 5.3: Proving the safety of crypto_verify function

force the provers to infer the range of the variable differentbits at each iteration, are also required to discharge all the safety proof obligations. Note that, the unfolded version of this algorithm does not require such annotations (without loops, termination is not an issue and no new states are introduced).

Establishing (termination-insensitive) security To apply Theorem 24, we establish security by first constructing a reified version of the program, and then performing a self-composition proof that it displays the required non-interference properties. The transformed program is created according to the rules described in Figure 5.3, and outputs a set of lists containing the relevant traces collected during the program's execution.

Recall that the list type introduced in the instrumented semantics of Section 5.3 is essentially an artifact to enable the application of our proof technique. They are not dynamic data structures offered by the underlying programming language, but rather constructions that may exist merely at the logical level. Furthermore, since the values of the constructed lists cannot influence the semantics of operations over other data types, they enable a more elegant formalisation and an easier justification of our theoretical results. Luckily, we can take advantage of a feature of Frama-c that enables the direct transposition of this logical data type onto code annotations: the ability to use *ghost code* in annotations enables us to include all the extra code introduced by our transformation as comments to the original program. Furthermore, using ghost code, we have the guarantee that the semantics of the original program are preserved, and cannot be affected by the values of said lists as required by our formalisation. This restriction is imposed as a necessary condition by the deductive verification tool.

In short, the fact that we do not require a concrete implementation of the list type is a central aspect to the practical side of our work. On one hand, it eliminates a potential gap between our theoretical and practical approaches. On the other hand, as noted in [87], if we could not adopt this strategy, the formal verification exercise would be rendered considerably more complex, and probably, out of reach of our framework.

In Listing 5.4 we show the result of applying the transformation in Figure 5.3 to the program in Listing 5.1. Note the declaration of C functions that allow the construction of the lists within ghost code. The semantics of these functions is axiomatised to capture the necessary list constructors. At the end of execution, the final state of the ghost list variables is essentially a logical term evidencing a sequence of cons operations. Our experience shows that this implementation is highly suitable for being passed down to automatic provers. To complete the verification exercise, we must establish that this

```
/*@ axiomatic list{ type list;
  @                 logic list null;
  @                 logic list cons(integer n, list s); }                  */

/*@ ghost int mem_control, mem_x, mem_y;
  @ axiomatic lmem{ logic list lmem_control{L} reads mem_control;
  @                 logic list lmem_x{L} reads mem_x;
  @                 logic list lmem_y{L} reads mem_y; }                    */

/*@ assigns mem_control;
  @ ensures lmem_control{Here} == cons(condition,lmem_control{Pre});        */
void append_control(int condition);
/*@ assigns mem_x;
  @ ensures lmem_x{Here} == cons(x,lmem_x{Pre});                            */
void append_x(int x);
/*@ assigns mem_y;
  @ ensures lmem_y{Here} == cons(y,lmem_y{Pre});                            */
void append_y(int y);

void crypto_verify(const unsigned char *x, const unsigned char *y) {
   int differentbits = 0, i = 0;

   //@ ghost append_control(i<16);
   while (i < 16) {
      differentbits |= x[i] ^ y[i]; //@ ghost append_x(i); ghost append_y(i);
      i++;
      //@ ghost append_control(i<16);
   }
   return (1 & ((differentbits - 1) >> 8)) - 1;
}
```

Listing 5.4: Transformed version of crypto_verify function

reified program indeed displays the non-interference property specified in Theorem 24. Here we directly apply, in a black-box way, the approach to performing proofs by self-composition presented in Chapter 4. Therefore, to deal with the loop structures, we annotate the programs with natural invariants and the associated lemmata. In particular, in order to enable the automatic discharge of all proof obligations, the lemma shown in Figure 5.5 needs to be included.

```
/*@ lemma eq_loop_pred{L1,L2,L3,L4}:
  @ \forall int i1,i2,i3,diffbits1,diffbits2,unsigned char *x,*y,*x1,*y1;
  @ \forall list l1_x,l2_x,l1_y,l2_y, l1_control, l2_control,l1_x1, l2_x1;
  @ \forall list l1_y1, l2_y1,l1_control1,l2_control1;
  @   l1_x == l1_x1 ==> l1_y == l1_y1 ==> l1_control == l1_control1 ==>
  @   loop_pred{L1,L2}(i1, i2, x, y, 0, diffbits1, l1_x, l2_x,
  @                    l1_y, l2_y, l1_control, l2_control) ==>
  @   loop_pred{L3,L4}(i1, i3, x1, y1, 0, diffbits2, l1_x1, l2_x1,
  @                    l1_y1, l2_y1, l1_control1, l2_control1) ==>
  @   i2 == i3 ==> l2_x == l2_x1 && l2_y == l2_y1 &&
  @     l2_control == l2_control1;
  @*/
```

Listing 5.5: Lemma of crypto_verify function

The rest of the Frama-c input can be found in Appendix B.2. Note that the preconditions include only the necessary restrictions to complete the proof, and need not refer to all the non-high parts of the initial state. As stated above, the discharging of the proof obligations generated by this example, bar the lemma presented above, was handled without assistance by the automatic provers targeted by Frama-c. Furthermore, although we have manually added these annotations, we emphasise that *all* of the annotations required for this verification exercise could have been generated automatically by a tool implementing the specification described in the previous chapter.

The only caveat to the automation potential of this approach, which is highlighted by this example, resides therefore in the justification of self-composition lemmas such as that presented above. As explained in the previous chapter, the proofs of the lemmas can be interactively done using some proof assistant such as Coq, or one can use the Coq library [7] developed for that purpose.

5.4.2 A more challenging verification example

We now discuss how our techniques allow us to deal with a wider class of programs than previous approaches along similar lines [87, 88]. In particular, we show how we deal with programs with complex control structures, including nested loops, and

```
static void mulmod(unsigned int h[17],const unsigned int r[17]) {
  unsigned int hr[17]; unsigned int i; unsigned int j; unsigned int u;
  for (i = 0;i < 17;++i) {
    u = 0;
    for (j = 0;j <= i;++j) u += h[j] * r[i - j];
    for (j = i + 1;j < 17;++j) u += 320 * h[j] * r[i + 17 - j];
    hr[i] = u;
  }
  for (i = 0;i < 17;++i) h[i] = hr[i];
  squeeze(h);
}
```

Listing 5.6: A snippet of the NaCl sources containing nested loops

also how we handle the verification of complete programs: self-contained components involving higher-level functions calling lower-level ones.

Listing 5.6 contains another snippet from the NaCl library implementation. This function carries out a specific modular multiplication operation. We have proved its adherence to the NaCl side-channel countermeasures using exactly the same approach as for the previous example. Intuitively, the natural invariant for the outer loop refers to the predicates specifying the natural invariants for the inner loops, as explained in the previous chapter. All loop invariants refer to the contents of the trace lists in a simple way, which is made possible by our formalisation of these lists directly using the ACSL logical types. The end result is that the proof obligations for this more elaborate example are also discharged automatically by the Frama-c backend provers. As before, the self-composition lemmas must be discharged interactively, with the assistance of some proof assistant such as Coq. This stands in contrast with the work presented in [87], in which nested loops are excluded.

Another important point in verifying the function in Listing 5.6 is that it is not a leaf function: it calls auxiliary function squeeze, which in turn is a leaf function. To handle function calls in NaCl , and because we have not explicitly captured these language constructions in our formalisation, we slightly abuse the compositionality theorem for our theoretical framework presented in Section 5.3. In particular, we rely on the fact that the sequential composition of two secure programs is itself secure, and simply verify that all functions, independently, comply with the NaCl security policies. We argue that this is acceptable because of the following facts about the NaCl implementation which allow us to conclude that function calls in NaCl cannot, in themselves, introduce dependencies:

- It relies only on the char and int data types and arrays thereof, and uses no dynamic memory allocation.

- The relative addresses of all called functions are fixed at compile time.
- Parameter passing in the NaCl library is extremely conservative: all parameters are passed on a call-by-value basis with the exception of byte arrays.
- In NaCl , the base addresses of byte arrays passed by reference are all fully determined at compile time, with constant offsets relative to the start addresses of the memory regions that the caller itself received.

An alternative approach, which we have also implemented, permits formally verifying programs relying on a slightly more flexible parameter passing convention. In particular, we could exclude programs that introduce dependencies when passing the base address of a memory region to a callee function using an offset that depends on a sensitive value. This implies enhancing the reification of a caller function to incorporate in its output traces the start addresses of all memory regions passed by reference to the callees. However, we leave for future work the formalisation and a full description of the implementation of our techniques for these more complex use cases.

5.5 Summary

In this chapter, we have shown how the techniques presented in the previous chapter can be applied to verify compliance to security policies aiming to reduce exposure to timing side-channel attacks. These properties are formulated in terms of *non-interference* and formalised using the self-composition technique [17]. We address the problem at the C source code level; our motivation is twofold:

- Our goal is to enable the formal verification of claims that were, until now, stated and checked in an informal way. Our solution is designed to respond to the concrete needs of cryptographers, by focusing on existing security policies and source-code that are used in real-world applications. In Section 5.3.3 we provide formal definitions of these security policies, so that their purpose and reach can be better understood. We also use these definitions to precisely justify the guarantees provided by our formal verification approach.
- Our solution is based once again on off-the-shelf formal verification tools and in the composition-based methodology introduced in the previous chapter. By

using existing tools, we are able to anchor the trust that may be deposited in our approach on a well-established and standard class of tools and techniques. At the same time, we demonstrate the applicability of existing technology to novel application areas, namely the formal verification of countermeasures against wider classes of side-channel attacks, and we demonstrate the potential for automating the verification of cryptographic software by self-composition, showing that we can tackle a wider class of programs than previous approaches in the same line.

Part II

CAOVerif : A deductive verification tool for CAO

Chapter 6

A deductive verification tool for CAO

The results achieved in the first part of this thesis demonstrated the great potential of the deductive verification techniques to verify security-relevant properties in cryptographic software implementations written in C. However, the level of automation that can be achieved using such implementations is not very high, partly because of the intrinsic characteristics of the C language. The CAO language, due to its simplicity, represents an opportunity to develop a domain-specific deductive verification tool, allowing the for the same verification techniques and hopefully with a higher degree of automation.

In this chapter we outline the implementation aspects of such deductive verification tool, named CAOVerif, focusing on how we tackle the interesting challenges presented by the CAO language. We introduce a formalisation in first-order logic of the rich mathematical data types that are used in cryptography. We base our presentation on real-world examples of CAO code, extracted from the open-source code of the NaCl cryptography library. We show how we fine tuned our tool to enable the fully automatic formal verification of simple properties (in particular safety properties), and also more ambitious proof goals (arising in general proofs of functional correctness).

6.1 The CAO programming language

The CAO language [66] was developed in the CACE project and allows for the practical description of cryptography-relevant programs. Unlike languages used in mathematical packages such as Magma or Maple, which allow high-level mathematical constructions to be described in their full generality, CAO is restricted to enabling the implementation of cryptography kernels (e.g. block ciphers and hash functions) and sequences of finite field arithmetics (e.g. for elliptic curve cryptography). CAO has been designed to allow the programmer to work over a syntax that is similar to that of C, while focusing on the implementation aspects that are most critical for security and efficiency. Ideally, CAO should allow the implementation of low-level cryptographic primitives in way which is close to the notation used in scientific papers and standards, so that correctness should follow almost directly from transcription. Conversely, one might expect that standards and scientific papers could specify cryptographic primitives in a language such as CAO.

The memory model of CAO is extremely simple: there is no dynamic memory allocation, evaluation of expressions produces no side effects, and the language has a call-by-value semantics. Furthermore, CAO does not support any input/output constructions, as it is targeted at implementing the core components in cryptographic libraries. On the other hand, the native types and operators in the language are highly expressive and tuned to the specific domain of cryptography. These features can be used by the CAO compiler to provide domain-specific analysis and optimisation when generating machine-level executable code.

A detailed specification of CAO can be found in [79, 10]. Next we will detail the most important features of this language. In Appendix C we include source code written in CAO, extracted from the NaCl cryptographic library.

6.1.1 CAO language overview

The syntax of CAO is detailed in [79, 10]. In a nutshell, as a C-like language, CAO includes conditionals and loops, as well as global variable declarations, function declarations and procedures. CAO expressions include among other things, integer arithmetic and comparison, and bit-wise and boolean operations. Perhaps worth noticing is the seq statement that permits expressing loop constructions where the number of iterations can be statically determined. Here, the iterator is an integer variable, seen as a read-only constant within the loop body. The syntax of expressions is also similar to that of C, although the set of types and operations is significantly different.

Type system The CAO type system includes a set of primitive types: int for arbitrary precision integers, bits[n] for bitstrings of finite length, mod[n] for rings of residue classes modulo an integer (intuitively, arithmetic modulo a composite integer, or a finite field of order n if the modulus is prime), bool for boolean values and void to represent the unit type. Derived types allow the programmer to define more complex abstractions. These include the product construction struct, the generic one-dimensional container vector[n] of τ, the algebraic notion of matrix, denoted matrix[i,j] of τ, and the construction of an extension to a finite field τ using a polynomial $p(X)$, denoted mod[τ <X> / $p(X)$] (intuitively, the finite field of order n^d can be implemented using arithmetic modulo an irreducible polynomial $p(X)$ over polynomials with coefficients in τ, where τ represents a finite field of order n and $p(X)$ has degree d).

CAO also allows the programmer to define new identifiers for the existing types in a similar way to the `typedef` construct in C. These new identifiers are known as *type synonyms* and their general purpose is to make programs clear and easier to read. An example of the definition of type synonyms is presented next.

```
typedef DES_KEY := unsigned bits[64];
def k : DES_KEY;
```

Observe that, afterwards one can declare a variable k using the new type name `DES_KEY`.

An implementation of a type-checker for CAO programs has been derived from the CAO type system formalisation [10] and in this thesis we assume that the input CAO programs that we are taking into account are type-safe (previously type-checked). We remark that type information includes in particular the concrete sizes of all container types, the moduli and polynomials in rings and finite fields, etc. Furthermore, the CAO type checker is able to reject all programs where incompatible type parameters are passed to an operator. For example, the size restrictions associated with matrix addition and multiplication are enforced by the type system. The same happens for operations involving bitstrings, rings and finite fields, where the type system checks that operator inputs have matching lengths, moduli, etc.

Operations Algebraic operators are overloaded so that expressions can include integer, ring/ finite-field and matrix operations. The ** operator represents the exponentiation operation, where the basis can be an integer or a value in a modular type, and the exponent must be a non-negative integer. The natural comparison operators, extended bitwise operators, boolean operators and a well-defined set of type conversion (cast) operators are also supported.

Bitstring, vector and matrix access operations are extended with the range selection (..) operator (these are also known as *slicing* operations). This operator is also more general than the C array: we can specify ranges of values for each index. For example, the following CAO code assigns the bits of x ranged between 0 and 3 to y.

```
def x : unsigned bits[8];
def y : unsigned bits[4];

y := x[0..3];
```

CAO also introduces the *concat* operator @. Vectors and bitstrings can be concatenated using the @ operator. The idea of @ is that it allows two operands to be "composed". For example, the following construction `y := x[0] @ x[1] @ x[2];` represents the application of concat operator and has the same meaning of `y := x[0..2];`.

The soundness of the type system has recently been established with respect to the semantics of CAO [9]. This result implies that a correctly typed CAO program can only give rise to a well-defined set of *trapped* errors. This notion has a direct bearing on the discussion of what *safety* means for CAO programs.

6.1.2 Safety in CAO

One of the requirements for CAOVerif is that safety verification should be feasible with minimum intervention from the end-user. Program safety in CAO has two dimensions: *memory safety* and *safety of arithmetic operations*. A program is said to be memory safe if it never fails at run-time by accessing an invalid memory address. Memory safety verification is not in general a trivial problem in languages with pointers and heap-based data structures, and indeed there exist dedicated verification tools for this task. However, for correctly typed CAO programs, this problem is reduced to making sure that all indices used in vector, bitstring and matrix index accesses are within the proper range.

The safety of arithmetic operations is more interesting. In CAO we have four algebraic types: *arbitrary precision integers*, *rings of residue classes* modulo a composite number, *finite fields*, and *matrices* thereof. The semantics of operators over these types is precisely given by the mathematical abstractions that they capture. This means that the concept of arithmetic overflow does not make sense in this context, and it leaves as candidate safety verification goals the possibility that such operators are not defined for certain inputs, and that such pathological cases might not be caught during type-checking.

Assuming that CAO program correctly type-checks, then matrix addition and multiplication are intrinsically safe (observe that there is only matrix addition and multiplication). The safety of integer operations includes the classic division-by-zero condition. Furthermore, the exponentiation operator over integers, rings and finite-fields is only defined for non-negative exponents (for usability reasons, programmers are required to denote multiplicative inverses explicitly through the division operator). Rings and finite fields pose other specific interesting problems, as they are not syntactically distinct CAO types. Take the following declarations[1].

```
def a : mod[13] := [4];          def b : mod[10] := [5];
def c : mod[13] := 1/a;          def d : mod[10] := 1/b;
```

All of these operations are safe, except for the initialization of d. The reason for this is that the multiplicative inverse modulo 10 is only defined for those integers in the range 1 to 9 that are coprime with 10. This means that, whenever a division occurs in the mod[n] type, one must also ensure that the divisor is coprime to the modulus.

When the modulus is a prime number, then the mod[n] type represents the finite field of size n. In this case, the previous problem reduces again to the division-by-zero case, as all non-zero elements have a multiplicative inverse. However, this observation does not help, unless there is a way to verify that the modulus is indeed a prime number. One way to do this, of course, is to allow the programmer to vouch for the primality of the modulus. We will return to this issue further in this thesis (at Section 6.3.5). Finally, a related problem arises when one considers the construction of extension fields. In this case, not only must one ensure that the underlying base type represents a finite field (which might not be the case for the mod[n] type) but also that the polynomial that is provided is irreducible in the corresponding ring of polynomials.

6.2 Implementation

The goal of the deductive verification tool for CAO is to allow developers to verify properties of CAO programs at the source code level. CAOVerif, follows the same approach used in other scenarios for general-purpose languages such as Java [67] and C [45]. In fact, part of the foreseen functionality for the tool is a direct adaptation of what is done for other languages. This includes, not only features such as the ability to check a program to be free of common programming errors and security vulnerabilities (such as

[1]Here the [·] syntax on literals distinguishes literals of modular types from integer literals.

those arising from incorrect calculations or indexing out of bounds), but also, and more generally, the power to reason about functional properties of programs expressed using logic assertions such as post-conditions. An implication of designing such a generic verification tool is that one must devise a way to enrich input programs with additional information. Typically this information takes the form of special annotations inserted in the source code as comments that follow a specific syntax.

CAO-SL CAO-SL, the annotation language that we envision being used in the CAO verification tool is mostly inspired by the behavioural interface specification language ACSL (already introduced in Chapter 3). One distinctive feature of this specification languages is the fact that the syntax of expressions is the same as (or a superset of) that used by the programming language, which makes the process of annotating a program much more intuitive than it would otherwise be.

The annotations in CAO-SL are embedded in comments (and thus ignored by the CAO compiler) using a special format recognised by the verification tool. The *logical expressions* used in annotations correspond to CAO expressions with additional constructs. CAO-SL includes the definition of *function contracts* with pre- and postconditions, statement annotations such as assertions and loop variants and invariants, and other annotations commonly used in specification languages. CAO-SL also allows for the declaration of new logic types and functions, as well as predicates and lemmas. A complete description of CAO-SL can be found in [66]. We remark that CAO-SL is rich enough to formalize arbitrary functional properties of CAO programs. More specifically, it allows the formalisation of security-relevant properties. Appendix C contains relevant examples of CAO programs annotated with CAO-SL specifications. Further in this chapter (see Section 6.4) we present how we can verify some of the security policies addressed in the previous chapters in cryptographic implementations written in CAO (extracted from the NaCl library).

6.2.1 Tool architecture

We first present our view of CAOVerif from a user's perspective. It is very important to clarify that, although the goal is to include as much automation as possible, the fact that we pursue a method which guarantees soundness with the highest level of assurance will very likely preclude a full fledged deductive verification tool.

The tool architecture itself fundamentally relies on the Jessie plug-in, which itself

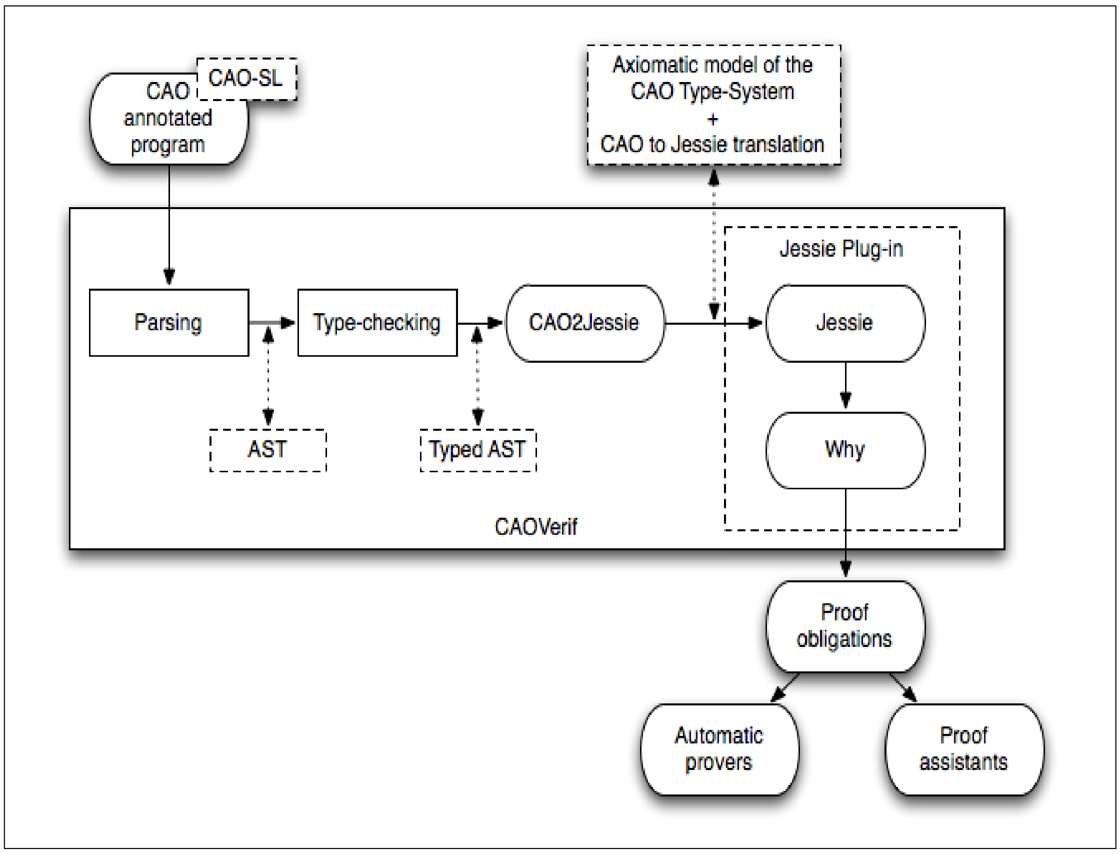

Figure 6.1: Tool architecture

uses Why as a back-end and is one of the components integrated into the Frama-c framework. This allow us to significantly reduce the tool development time and effort. Jessie enables reasoning about typical imperative programs, and it is equipped with a first-order logic mechanism, which facilitates the design of new models and extensions. In particular, it is possible to use this feature to define in Jessie a model of the domain-specific types and memory model of CAO. This means that an annotated CAO program can be translated into an annotated Jessie program and, from this point on, our verification tool can rely totally on the functionality of Jessie and Why. The diagram in Figure 6.1 outlines the architecture of our tool. We remark that in the next chapter (Chapter 7) we reason about the correctness of this approach.

- 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 errors and typing errors. After obtaining a well-typed Abstract Syntax Tree (AST) we then translate it into Jessie input language, using the front-end component for CAO (`Cao2Jessie`).

- 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.
- The proof obligations are generated by running the Jessie plug-in, which uses Why as a back-end, on the output of `Cao2Jessie` tool. The proof obligations can then be checked by some existent automatic prover or proof assistant. Using this implementation strategy, we inherit the advantages of separating the design of the tool into modules.

One advantage of this modular architecture is that it allows the enrichment of the annotation language without the necessity of changing the VCGen input language, and vice-versa. Conversely, the VCGen mechanism can be changed without modifying the specification language (in particular it can be interfaced with additional proof assistants and proof tools).

6.2.2 Strategy

In order to better illustrate our approach to designing a VCGen for CAO taking advantage of an existing *generic* VCGen, we introduce a very simple example. Consider the definition of the VCGen introduced in Chapter 2 of the *While*^{C} language, that corresponds to an abstraction of the CAO language, and how one deals with *safety*. The weakest precondition of the array assignment operation resembles the following

$$\mathsf{wp}(\mathsf{a}[e_1] := e_2, \psi) = \mathsf{safe}(\mathtt{a}[e_1]) \wedge \mathsf{safe}(\mathsf{e}_2) \wedge \psi[\mathsf{upd}(\mathsf{a}, e_1, e_2)/\mathtt{a}]$$

where $\mathsf{safe}(\mathtt{a}[e_1]) = \mathsf{safe}(\mathsf{e}_1) \wedge 0 \leq e_1 < \mathsf{len}(\mathsf{a})$ and, $\mathsf{safe}(e_1)$ and $\mathsf{safe}(e_2)$ impose that the evaluation of e_1 and e_2 will not produce arithmetic errors.

Rather than implementing the VCGen from scratch, one alternative possibility is to construct the VCGen for *While*^{C} on top of the *While** VCGen[2] (also introduced in Chapter 2). Recall that *While** is an abstraction of the Jessie language where the array type is introduced as a logical type, for which a model is given. Leaving safety

[2] *While*^{C} annotated programs are translated into *While** input language, and the *While** VCGen is used to generate the proof obligations.

considerations aside for the moment, the array assignment command could be translated as follows, where $\langle\cdot\rangle$ denotes the translation of both program instructions and expressions:

$$\langle \mathsf{a}[e_1] := e_2\rangle \;\equiv\; \mathsf{a} = \mathrm{set}(\mathsf{a}, e_1, e_2)$$

In the concrete case of CAO and Jessie, one can rely on the Jessie VCGen to follow this approach for the entire CAO language. There is an overlap between the CAO language and the Jessie input language that enables a direct translation of many language constructions. Furthermore, for each CAO type that is not supported by Jessie we are able to declare a set of logical functions, and write a theory that creates a first-order model of the type. Then this enables us to translate arbitrary annotated CAO programs into suitable programs of the Jessie input language.

Let us now turn back to the example above, to see how we deal with safety conditions using *While**'s *assert* clause to force the generation of arbitrary proof obligations. Safety conditions for arrays can be generated by translation the assignment instruction as follows:

$$\langle \mathsf{a}[e_1] := e_2\rangle \;\equiv\; \textbf{assert}\; 0 \le \langle e_1\rangle < \mathsf{len}(\mathsf{a}) \;\wedge\; \phi; \quad \mathsf{a} = \mathrm{set}(\mathsf{a}, \langle e_1\rangle, \langle e_2\rangle)$$

where ϕ corresponds to the conjunction of the conditions necessary to guarantee the safe evaluation of e_1 and e_2. Of course, in CAO we have to deal with data types that are considerably more sophisticated than arrays. Yet, the general pattern followed in the implementation of our tool is the same. The introduction of each new type implies the introduction of a new theory, including the definition of logical functions together with axioms to model their behavior. Some lemmas and predicates may also be introduced to facilitate the process of proving goals. The correctness of this approach will be addressed in Chapter 7, where we discuss how one can establish the soundness of a verification tool developed in such a way.

6.2.3 Emphasis on automation

The fact that Jessie relies on the Why VCGen, which is a *multi-prover* tool, means that it is possible to export verification conditions to a large number of different proof tools, from SMT-solvers to the Coq interactive proof assistant. The typical workflow is to first discharge “easy” VCs using an automatic prover, and then interactively handle the remaining conditions. Our translation enables varying degrees of automation, depending on the complexity of the verification goals. As is the case with VCGens for other realistic languages, one expects safety conditions to be proved with a high degree of automation, whereas a lower degree is acceptable for other functional properties.

The degree of automation that we can achieve in verifying the safety of CAO programs is quite high. We are able, for example, to carry out without user intervention the safety verification of the entire CAO implementation of crypto_scalar_mult function included in Appendix C, which includes heavy use of finite field, vector and matrix operations, across several dependent functions. We are also able to automatically deal with surprisingly intricate proofs of functional correctness with only minor intervention from the user in interactively discharging proof obligations.

6.3 CAO to Jessie translation

In this section we will resort to snippets of CAO code to describe the most interesting parts of the CAO to Jessie translation carried out by our verification tool, which essentially correspond to the rich cryptography-specific data types that are available in CAO. In other words, we will focus on the way in which we handle the parts of the CAO language (including the extension to CAO-SL) that do not directly map to constructions in the Jessie input language, leaving out the standard imperative constructions supported by both languages, the CAO types that directly map to Jessie native types, and the translation of annotations, which is also direct (in Appendix D we detail the remaining translations). In the following, $\langle \mathsf{x} \rangle$ denotes the translation of a part of the input CAO program x into Jessie. Here x can denote any part of the input AST, e.g. a full program, a type declaration, an expression, etc.

Figure 6.2 gives an overview of how CAO type declarations are translated into Jessie type declarations. Some CAO primitive types are translated to Jessie primitive types, namely int, bool and void. This means that, for these CAO data types, we directly benefit from the models already provided by the Jessie plug-in for reasoning about the target Jessie native types.

The remaining CAO types are mapped into newly declared Jessie logic types. Note that, for parametrised data types such as mod[n], the target type in Jessie is named so as to explicitly capture the type parameter. This also explains why we use the translation

$$
\begin{array}{rr}
\langle \mathsf{int} \rangle = \mathsf{integer} & \langle \mathsf{bits[n]} \rangle = \mathsf{bits} \\
\langle \mathsf{bool} \rangle = \mathsf{boolean} & \langle \mathsf{matrix[n_1,n_2]\ of\ } \tau \rangle = \mathsf{matrix\_}\langle \tau \rangle \\
\langle \mathsf{void} \rangle = \mathsf{unit} & \langle \mathsf{mod[}\tau \mathsf{\ <X>\ /\ p(X)]} \rangle = \mathsf{field\_}\langle \tau \rangle\mathsf{\_}\langle \mathsf{f(X)} \rangle \\
\langle \mathsf{mod[n]} \rangle = \mathsf{mod\_}n & \langle \mathsf{vector[n_1]\ of\ } \tau \rangle = \mathsf{vector\_}\langle \tau \rangle
\end{array}
$$

Figure 6.2: Type translation.

operation recursively in Figure 6.2. In the following, we discuss how we enrich the generated Jessie input file with logic models that partially capture the semantics of the translated CAO types, in order to enable both automatic and interactive reasoning about the input CAO program.

6.3.1 Container types

The container types in CAO include the vector[] of, matrix[] of and bits types. The get and set operations on these types are modeled in Jessie using exactly the second approach that we described in the example in the previous section. The only caveat is that they are generalized to two dimensions in the case of matrices, and that we set Jessie type bool as the content type in the case of bitstrings. For instance, the axiomatic model of the vector[] of τ type, includes the definition the following logical functions, together with the axioms to model their behaviour.

$$
\begin{aligned}
\mathsf{vector\_}\langle\tau\rangle\mathsf{\_get} &: \mathsf{vector\_}\langle\tau\rangle \rightarrow \mathsf{integer} \rightarrow \langle\tau\rangle \\
\mathsf{vector\_}\langle\tau\rangle\mathsf{\_set} &: \mathsf{vector\_}\langle\tau\rangle \rightarrow \mathsf{integer} \rightarrow \langle\tau\rangle \rightarrow \mathsf{vector\_}\langle\tau\rangle
\end{aligned}
$$

$$
\begin{array}{l}
\forall\ \mathsf{vector\_}\langle\tau\rangle\ v,\ \mathsf{integer}\ i,\ \tau\ x.\ \mathsf{vector\_}\langle\tau\rangle\mathsf{\_get}(\mathsf{vector\_}\langle\tau\rangle\mathsf{\_set}(v,i,x),i) == x \\
\forall\ \mathsf{vector\_}\langle\tau\rangle\ v,\ \mathsf{integer}\ i,j,\ \tau\ x.\ i\ !=j \Longrightarrow \\
\qquad\qquad \mathsf{vector\_}\langle\tau\rangle\mathsf{\_get}(\mathsf{vector\_}\langle\tau\rangle\mathsf{\_set}(v,i,x),j) == \mathsf{vector\_}\langle\tau\rangle\mathsf{\_get(v,j)}
\end{array}
$$

Additionally, CAO includes elaborate operators to deal with these container types that are fine-tuned to the implementation of cryptographic algorithms, namely symmetric primitives such as block ciphers and hash functions. As an example, consider the next snippet from a CAO implementation of the Advanced Encryption Standard (AES) [1] block cipher (the full implementation in CAO can be found in Appendix E).

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

What we have here is a sequence of rotation (|>) operations applied to the i^{th} row of a 4×4 matrix s. The way in which this is expressed in CAO takes advantage of the range selection operator (..) that returns a value of the corresponding container type, with the same contents as the original one, but with appropriate dimensions. Here, this operator is used to select an entire row in the matrix, which is cast into the correct vector type (here the RowV type denotes a vector of size 4) in order to be rotated. The result is then cast back to the correct matrix type that can be assigned to the original row slice in matrix r.

Our first-order formalisation of container types deals with shift, rotate, range selection, range assignment and concatenation (@) operators in container types using a pattern that relies on two logic functions (shift and blit). We present the case of the vector type. The model assumes that a vector has infinite length, i.e., it has a start position, but it is represented as an unbounded memory block. The only exception to this rule is the extensional equality operator (==), where translation explicitly refers to the range of valid positions over which equality should hold. We emphasize that this part of the model deals only with the functionality of these operators: safety is handled separately by introducing appropriate assertions, as will be seen in Section 6.3.5.

Intuitively, the shift logic function takes as input a vector of arbitrary length, starting in position 0, and produces the vector that starts at position i. The blit logic function 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. The behaviour of these logic functions is modeled by the declarations and axioms given in Figure 6.3.

Range selection Given a CAO variable μ of type $\mathsf{vector}[n]$ of τ, the CAO range selection operation is modeled in Jessie as follows:

$$\langle\ \mu[\mathsf{i..j}]\ \rangle\ \equiv\ \textbf{let}\ x_1 = \langle \mathsf{i} \rangle\ \textbf{in}\ (\ \textbf{let}\ x_2 = \langle \mathsf{j} \rangle\ \textbf{in}$$
$$\textbf{assert}\ (0 \le x_1 < n)\ \&\&\ (0 \le x_2 < n)\ \&\&\ (x_1 \le x_2);\ \mathsf{shift}(\langle \mu \rangle, x_1))$$

$$\text{blit\_vector\_}\langle\tau\rangle \;:\; \text{vector\_}\langle\tau\rangle \rightarrow \text{vector\_}\langle\tau\rangle \rightarrow \text{integer} \rightarrow \text{integer} \rightarrow \text{vector\_}\langle\tau\rangle$$
$$\text{shift\_vector\_}\langle\tau\rangle \;:\; \text{vector\_}\langle\tau\rangle \rightarrow \text{integer} \rightarrow \text{vector\_}\langle\tau\rangle$$

$$\forall v, ofs, i.\ \text{get\_vector\_}\langle\tau\rangle(\text{shift\_vector\_}\langle\tau\rangle(v, ofs), i) = \text{get\_vector\_}\langle\tau\rangle(v, (ofs + i))$$

$$\forall src, dest, ofs, len, i.\ ofs \leq i < (ofs + len) \implies$$
$$\text{get\_vector\_}\langle\tau\rangle(\text{blit\_vector\_}\langle\tau\rangle(src, dst, ofs, len), i) = \text{get\_vector\_}\langle\tau\rangle(src, i - ofs)$$

$$\forall src, dest, ofs, len, i.\ i < ofs \ \vee\ i \geq (ofs + len) \implies$$
$$\text{get\_vector\_}\langle\tau\rangle(\text{blit\_vector\_}\langle\tau\rangle(src, dst, ofs, len), i) = \text{get\_vector\_}\langle\tau\rangle(dst, i)$$

Figure 6.3: Declarations and axioms for vector types.

where i and j are integer expressions. We remark that although the translation disregards the upper bound j in the call to shift, the type-checking phase has ensured that the range selection operation $\mu[i..j]$ with μ of type vector[n] of τ, returns type vector[$j - i + 1$] of τ, thus implicitly taking that upper bound into account. Furthermore, all future accesses to the resulting vector will be checked for safety within the valid bounds prescribed by the associated data type.

Range assignment Assigning to a region in a vector is modeled directly using the blit function.

$$\begin{aligned}\langle \mu_1[\text{i..j}] := \mu_2 \rangle \equiv \langle \mu_1 \rangle = \ & \textbf{let}\ x_1 = \langle \text{i} \rangle\ \textbf{in}\ (\ \textbf{let}\ x_2 = \langle \text{j} \rangle\ \textbf{in} \\ & \textbf{assert}\ (0 \leq x_1 < n)\ \&\&\ (0 \leq x_2 < n)\ \&\&\ (x_1 \leq x_2); \\ & \text{blit}(\langle \mu_2 \rangle, \langle \mu_1 \rangle, x_1, x_2 - x_1 + 1))\end{aligned}$$

Concatenation Consider the CAO variables μ_1 and μ_2 of types vector[n_1] of τ and vector[n_2] of τ respectively. The concatenation of vectors μ_1 and μ_2 can also be captured using the blit function.

$$\langle \mu_1\ @\ \mu_2 \rangle \equiv \text{blit}(\langle \mu_2 \rangle, \langle \mu_1 \rangle, n_1, n_2)$$

The intuition behind this definition is that concatenation can be seen as a range assignment operation, where μ_2 is assigned to the region of μ_1 that starts at position n_1 (recall that in the model vectors are assumed to have infinite length).

Initialisation Vectors initialisation is done using the function any_vector_$\langle\tau\rangle$ which does not have any input values, but produces as output a value of type vector_$\langle\tau\rangle$.

$$\langle \textbf{def}\ \mathsf{v : vector[n]\ of}\ \tau\rangle \ \equiv\ \textbf{var}\ \mathsf{vector\_}\langle\tau\rangle\ \mathsf{v = any\_vector\_}\langle\tau\rangle()$$

Matrices Our model of matrices is a direct generalization of the above strategy to the 2-dimensional case. However, our model of matrices must also account for the fact that the matrix type in CAO is an algebraic type that supports addition and multiplication operations (indeed this is why in CAO you can only define matrices whose contents are themselves algebraic types).

The formalisation of matrices in first-order logic includes the matrix addition and multiplication arithmetic operations as logic functions

$$\mathsf{matrix\_}\langle\tau\rangle\mathsf{\_add}, \mathsf{matrix\_}\langle\tau\rangle\mathsf{\_mult} : \mathsf{matrix\_}\langle\tau\rangle \to \mathsf{matrix\_}\langle\tau\rangle \to \mathsf{matrix\_}\langle\tau\rangle$$

The functionality of the addition operator is modeled using the following axiom:

Axiom 1. *Let A and B be matrices of dimensions $m \times n$, and a_{ij} and b_{ij} the elements in the i^{th} row and j^{th} column of A and B, respectively. Then, $\forall\ j, i.\ (A + B)_{ij} = a_{ij} + b_{ij}$.*

An equivalent axiom for matrix multiplication was not introduced because, for each possible base type, we would need the (higher-order) logic formalization of the mathematical (iterative) sequence summation operator Σ.

The translation of expressions with arithmetic operations of type matrix[n_1,n_2] of τ is therefore the following:

$$\begin{aligned}\langle\ \mu_1\ +\ \mu_2\ \rangle &= \mathsf{matrix\_}\langle\tau\rangle\mathsf{\_add}(\langle\mu_1\rangle, \langle\mu_2\rangle)\\ \langle\ \mu_1\ *\ \mu_2\ \rangle &= \mathsf{matrix\_}\langle\tau\rangle\mathsf{\_mult}(\langle\mu_1\rangle, \langle\mu_2\rangle).\end{aligned}$$

Shift and rotate To present the shift and rotate operations in a more intuitive way, we will turn to the bits type. Both operations are modeled using the blit function. The rotate operations are commonly known as circular shifts. A downwards circular shift by 1 is defined as a permutation of the entries in a tuple where the last element becomes the first element and all the other elements are down-shifted one position. Conversely, in an upwards circular shift, the first element becomes the last element and all the others are shifted up. As an example, consider the bits literal: 0b1101001. The internal representation of bits in our model stores the least significant bit (the right-most bit) in the 0-th position. This means that an upwards (resp. downwards) rotate corresponds to the intuitive interpretation of a left (resp. right) rotation. An example of a down rotate is therefore

$$0b1101001 \mid> 3 \; = \; 0b\underline{001}1101$$

and an example of an up rotate is

$$0b1101001 <\mid 3 \; = \; 0b1001\underline{110}.$$

In our model, for a CAO expression e of type vector[n] of τ or bits[n], we have:

$$\begin{aligned}
\langle \mathsf{e} <\mid \mathsf{i} \rangle &\equiv \langle \mathsf{e[n-i\,..\,n-1]\;@\;e[0\,..\,n-i-1]} \rangle \equiv \mathsf{blit(shift(}\langle \mathsf{e} \rangle, 0), \mathsf{shift}(\langle \mathsf{e} \rangle, n-i), i, n-i)\\
\langle \mathsf{e} \mid> \mathsf{i} \rangle &\equiv \langle \mathsf{e[i\,..\,n-1]\;@\;e[0\,..\,i-1]} \rangle \equiv \mathsf{blit(shift(}\langle \mathsf{e} \rangle, 0), \mathsf{shift}(\langle \mathsf{e} \rangle, i), n-i, i)
\end{aligned}$$

where i is a constant of type *int*. The intuition is that rotations can be seen as concatenations of the appropriate sub-regions, which in turn are modeled using the *blit* function.

Logical shifts are handled in a similar way, but resorting to bits_null_vector (a logical variable representing the all-zeroes bits value) to fill in the positions left vacant by the operation, i.e.,

$$\begin{aligned}
\langle \mathsf{e} \ll \mathsf{i} \rangle &\equiv \mathsf{blit(shift(}e, 0), \mathsf{bits\_null\_vector}, i, n-i)\\
\langle \mathsf{e} \gg \mathsf{i} \rangle &\equiv \mathsf{blit(bits\_null\_vector, shift(}\langle \mathsf{e} \rangle, i), n-i, i)
\end{aligned}$$

To model the behaviour of the bits_null_vector logical variable we include the following axiom in the bits type theory:

$$\forall \; \mathsf{integer}\; j.\; \mathsf{bits\_get(bits\_null\_vector,}\; j) == \mathsf{false;}$$

We remark that our model of the operations over bitstrings is complete, and therefore allows us to deduce the natural properties of bitstring operations. Furthermore, surprisingly complex properties can be derived automatically. Consider, for example, the bistring rotation operation and the property that rotating n times a bitstring of length n in the same direction yields the original bitstring:

$$\forall i.\;\; 0 \leq i < n \implies \langle \mathsf{e[i]} \rangle == \langle \mathsf{(e \mid> n)[i]} \rangle$$

$$\forall i.\ 0 \leq i < n \implies \langle \text{e[i]} \rangle == \langle \text{(e <| n)[i]} \rangle$$

Or, more generally, for a bitstring of length $n_1 + n_2$,

$$\forall i.\ 0 \leq i < (n_1 + n_2) \implies \langle \text{(e <| } \mathsf{n}_1\text{)[i]} \rangle == \langle \text{(e |> } \mathsf{n}_2\text{)[i]} \rangle$$

Our model enables proving these properties automatically using, e.g., `Alt-Ergo`.

Bits literal representation CAO bits representation specifies that the least significative bit is the right most bit, and the most significative is the left most bit. For instance, the literal $0b1010$ is represented as follows:

$$\begin{aligned} 0b1011 \ \equiv \ & \text{bits\_set}(\mathit{bits\_set}(\mathit{bits\_set}(\text{bits\_set}(\\ & \quad \text{bits\_null\_vector}, 0, \mathit{true}), 1, \mathit{true}), 2, \mathit{false}), 3, \mathit{true}). \end{aligned}$$

Equality and inequality To compare bitstrings and matrices we include in each model a logical predicate together with the corresponding axiom. For instance, the predicate bits_eq, used to compare two bitstrings of length l, is axiomatised as follows:

$$\begin{aligned} & \forall \text{ bits } b_1,\ b_2.\ \forall \text{ integer } l. \\ & \quad \text{bits\_eq}(b_1,\ b_2,\ l) \equiv (\forall \text{ integer } i.\ 0 <= i < l \Rightarrow \text{bits\_get}(b_1, i) == \text{bits\_get}(b_2, i)). \end{aligned}$$

However, Jessie does not allow logical predicates to appear in the standard language constructions (such as conditional branches and loops). For this reason, we also associate to each predicate a Jessie function that is used in the translation of CAO constructions (those that are not annotations) which include comparisons. Each function returns a boolean value and indicates whether the logical predicate holds or not. For example, the Jessie definition of the function used to compare two bitstrings is as follows:

```
boolean bits_eq_param(bits x, bits y, integer l)
behavior default:
  ensures (if \result then bits_eq(x,y,l) else (! bits_eq(x,y,l)));;
```

Bitwise operations We complete this section with a brief description of how bitwise operations are handled in our model, as these are of critical importance in cryptographic applications. Here we greatly benefit from the design of the CAO language, where the classic ambivalence between integers and their bit-level representations (that exists in the C int type) is eliminated by introducing the bits type. Indeed, CAO programmers can freely use bitstrings of any size, and convert these to and from the type int that represents the mathematical type $\mathbb{Z}$. A very simple model of bitstrings based on vectors of bits (boolean values) can be used, although things get more complicated when we need to deal with type conversions. The Jessie model of bitwise operations on bits is based on the following logic functions, which are axiomatized in the obvious way:

$$\text{bits\_bitwise\_xor} : \text{bits} \to \text{bits} \to \text{bits} \qquad \text{bits\_bitwise\_and} : \text{bits} \to \text{bits} \to \text{bits}$$
$$\text{bits\_bitwise\_or} : \text{bits} \to \text{bits} \to \text{bits} \qquad \text{bits\_bitwise\_neg} : \text{bits} \to \text{bits}$$

CAO bitwise operations are translated as:

$$\langle \mathrm{e}_1 \oplus \mathrm{e}_2 \rangle \equiv \text{bits\_bitwise\_}\langle\oplus\rangle(\langle \mathrm{e}_1\rangle, \langle \mathrm{e}_2\rangle) \qquad \langle !\ \mathrm{e}\rangle \equiv \text{bits\_bitwise\_neg}(\langle \mathrm{e}\rangle)$$

where $\oplus \in \{|, \&, \hat{}\}$ and μ_1 and μ_2 are expressions of type bits$[n]$.

6.3.2 Rings, fields and extension fields

Residue classes modulo n The mod$[n]$ type is an algebraic type. For $n \in \mathbb{N}$, it corresponds to the algebraic ring $\mathbb{Z}_n$. Moreover if n is prime, then mod$[n]$ permits programmers to take full advantage of the fact that $\mathbb{Z}_n$ is a field.

More in detail, the Jessie model for the mod$[n]$ type is based on the congruence relation defined by n over the integers. For a positive integer n, two integers a and b are said to be *congruent modulo n* if $a - b$ is an integer multiple of n, and this is denoted by $a \equiv b\ (mod\ n)$.

For any integer a, the corresponding equivalence class modulo n is denoted by $[a]$, and it corresponds to the set $a + n\mathbb{Z}$, where $n\mathbb{Z}$ is the set of multiples of n. For all integers a, the unique value r satisfying $a = nq + r \wedge 0 \leq r < n$ (for some integer q) is called the *least residue* of a modulo n. The set $\{0, 1, ..., n-1\}$ is therefore called the set of least residues modulo n. Each residue class modulo n is represented by a least residue modulo n.

The model of mod$[n]$ starts with the definition of the logic type mod_n, which intuitively is inhabited by the residue classes modulo n. This type is equipped with logic functions that convert to and from the Jessie integer type, as well as the mapping that results from their composition.

$$\begin{aligned}\mathsf{int\_of\_mod\_}n &: \mathsf{mod\_}n \to \mathsf{integer}\\ \mathsf{mod\_}n\mathsf{\_of\_int} &: \mathsf{integer} \to \mathsf{mod\_}n\\ \mathsf{mod\_}n &: \mathsf{integer} \to \mathsf{integer}\end{aligned}$$

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. The mod_n function represents the composition of the previous two, and associates to each $a \in \mathbb{Z}$ the least residue $r \in \mathbb{Z}$ of $[a]$. The model includes a set of axioms for the following mathematical properties of these functions:

$$\begin{aligned}&\forall x.\ 0 \le \mathsf{int\_of\_mod\_}n(x) \le n-1\\ &\forall x.\ 0 \le x \le n-1 \implies \mathsf{mod\_}n(x) = x\\ &\forall x.\ x \ge n \implies \mathsf{mod\_}n(x) = \mathsf{mod\_}n(x-n)\\ &\forall x.\ x < 0 \implies \mathsf{mod\_}n(x) = \mathsf{mod\_}n(x+n)\\ &\forall x.\ \mathsf{mod\_}n(\mathsf{int\_of\_mod\_}n(\mathsf{mod\_}n\mathsf{\_of\_int}(x))) = \mathsf{mod\_}n(x)\end{aligned}$$

Equipped with these functions we can base our entire model of integers modulo n on the theory of integers included in Jessie, which permits taking advantage of built-in arithmetic supported by many automatic provers.

The Jessie translation of arithmetic operations involving expressions of type mod[n] is based on the homomorphisms declared above. First, int_of_mod_n is used to get the least residues of the equivalence classes involved in the arithmetic operation, which is then carried out over the integers. Finally, we apply mod_n_of_int to the result to recover the equivalence class that represents the result. Hence, the translation of arithmetic operations on type mod[n] is given as follows, for $op \in \{+, -, *\}$.

$$\begin{aligned}\langle \mathsf{e}_1\ op\ \mathsf{e}_2\rangle &\equiv \mathsf{mod\_}n\mathsf{\_of\_int}(\mathsf{int\_of\_mod\_}n(\langle \mathsf{e}_1\rangle)\ op_{\mathsf{integer}}\ \mathsf{int\_of\_mod\_}n(\langle \mathsf{e}_2\rangle))\\ \langle \mathsf{e}_1 ** \mathsf{e}_2\rangle &\equiv \textbf{let } x = \langle \mathsf{e}_2\rangle \textbf{ in assert } x \ge 0;\\ &\quad\ \mathsf{mod\_}n\mathsf{\_of\_int}(\mathsf{int\_of\_mod\_}n(\langle \mathsf{e}_1\rangle){**}_{\mathsf{integer}}\ x)\\ \langle \mathsf{e}_1\ /\ \mathsf{e}_2\rangle &\equiv \textbf{let } x = \mathsf{int\_of\_mod\_}n(\langle \mathsf{e}_2\rangle) \textbf{ in assert } \mathsf{gcd}(x,n) = 1;\\ &\quad\ \mathsf{mod\_}n\mathsf{\_of\_int}(\mathsf{int\_of\_mod\_}n(\langle \mathsf{e}_1\rangle)\ {*}_{\mathsf{integer}}\ \mathsf{inv\_mod}(x,n))\end{aligned}$$

Exponentiation is translated so as to ensure that verification guarantees that the exponent is nonnegative, which would otherwise result in an error according to the semantics of the language. Also note the special case of division. This is justified because the semantics of division modulo n is not the same as integer division. Firstly, one must express the correct semantics, which we do by introducing the logical function $\mathsf{inv\_mod}(x, n)$. Simple properties involving operations with this function, which are used to automatically discharge some proof obligations, are axiomatized as:

$$\begin{aligned}
&\forall x.\ \mathsf{gcd}(\mathsf{int\_of\_mod\_}n(x)), n) = 1 \implies \\
&\qquad \mathsf{mod\_}n(\mathsf{int\_of\_mod\_}n(x) *_{\mathsf{integer}} \mathsf{inv\_mod}(\mathsf{int\_of\_mod\_}n(x), n)) = \mathsf{mod\_}n(1) \\
&\forall x, y.\ \mathsf{mod\_}n(\mathsf{int\_of\_mod\_}n(x) *_{\mathsf{integer}} y) = \mathsf{mod\_}n(1) \implies \\
&\qquad \mathsf{inv\_mod}(\mathsf{int\_of\_mod\_}n(x), n) = \mathsf{mod\_}n(y)
\end{aligned}$$

Secondly, in the division case, one must generate a proof obligation for the safety condition that CAO programs should not perform undefined divisions. This property is trivially true if the divisor is in the range $1 \ldots n-1$ and the number n is prime. Hence we add the following axiom to our model, to automatically handle these trivial cases.

$$\forall x, n.\quad \mathsf{is\_prime}(n) \wedge (0 < x < n) \implies \mathsf{gcd}(x, n) = 1$$

where $\mathsf{is\_prime} : \mathsf{integer} \rightarrow \mathsf{boolean}$ is a predicate to check if an integer number is prime, and $\mathsf{gcd} : \mathsf{integer} \rightarrow \mathsf{integer} \rightarrow \mathsf{integer}$ is a logic function that calculates the greatest common divisor of two integer numbers. Note that is_prime and gcd are neither directly defined nor axiomatized, but the programmer can explicitly assert that some n is prime through a CAO-SL annotation. This enables automatically discharging safety assertions using gcd.

Extension fields Consider the following type declarations taken from the same AES implementation referred above:

```
typedef GF2  := mod[2];
typedef GF2N := mod[GF2<X> / X**8+X**4+X**3+X+1];
typedef GF2C := mod[GF2N<Y> / Y**4+1];
```

Take the first 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 $\mathbb{Z}_n[X]$, 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.

The theory of extension fields of this form begins with the definition of a logic type ring_mod_n that represents the ring of polynomials over the base type mod[n] and logic functions to construct the elements of the ring and the addition operation that permits combining them.

$$\mathsf{ring\_mod\_}n\mathsf{\_monomial} : \mathsf{mod\_}n \rightarrow \mathsf{integer} \rightarrow \mathsf{ring\_mod\_}n$$
$$\mathsf{ring\_mod\_}n\mathsf{\_add} : \mathsf{ring\_mod\_}n \rightarrow \mathsf{ring\_mod\_}n \rightarrow \mathsf{ring\_mod\_}n$$

Our model explicitly captures the fact that elements of this ring are polynomials, which in turn can be defined as an addition of monomials. The reason for this is that the CAO literal that corresponds to the irreducible polynomial field_mod_n_poly_$f(x)$_generator used to construct these types can then be represented in our logical model. A monomial can be represented by its coefficient (which is an element of mod[n]) and its degree (an integer).

Arithmetic operations over the polynomial ring are not included in the model, as they do not exist in CAO. Indeed our model is purposefully incomplete because we do not intend to use automatic theorem provers on verification conditions involving arbitrary extension field algebra. The goal is to use a specific interactive proof assistants, namely Coq, to prove these kinds of properties, relying on existing libraries (e.g. SSReflect[3]) that provide theories for abstract algebra (fields, polynomials, etc).

The model is completed with definitions for type field_mod_n_poly_$f(x)$ and the corresponding arithmetic operations. The Jessie translation of the arithmetic operations defined for type mod[mod[n] <X> / $p(X)$] is then a direct one:

$$\begin{aligned}
\langle \mathsf{e}_1\ op\ \mathsf{e}_2\rangle &\equiv \langle \mathsf{e}_1\rangle\ op_{\mathsf{field\_mod\_}n\mathsf{\_poly\_}f(x)}\ \langle \mathsf{e}_2\rangle \\
\langle \mathsf{e}_1\ {**}\ \mathsf{e}_2\rangle &\equiv \textbf{let}\ x = \langle \mathsf{e}_2\rangle\ \textbf{in assert}\ x \geq 0; \\
&\qquad \langle \mathsf{e}_1\rangle\ {**}_{\mathsf{field\_mod\_}n\mathsf{\_poly\_}f(X)}\ x \\
\langle \mathsf{e}_1\ /\ \mathsf{e}_2\rangle &\equiv \textbf{let}\ x = \langle \mathsf{e}_2\rangle\ \textbf{in assert}\ x \neq 0_{\mathsf{field\_mod\_}n\mathsf{\_poly\_}f(X)}; \\
&\qquad \langle \mathsf{e}_1\rangle\ div_{\mathsf{field\_mod\_}n\mathsf{\_poly\_}f(X)}\ x
\end{aligned}$$

[3]http://www.msr-inria.inria.fr/Projects/math-components

where $op \in \{+, -, *\}$. Note that there is also a special case for exponentiation and division. This ensures that a safety proof obligation is generated that checks if the exponent is nonnegative (an integer) and that the divisor is different from zero, respectively.

A set of axioms that describe basic properties of these operators has been added to the model in order to increase the degree of automation provided by our tool. The goal here is that, given that there is no integrated support for this sort of mathematical construction in the automatic provers interfaced with Jessie, some simple properties can be captured in first-order logic that permit dealing with trivial steps, e.g. cancellation rules. The following axioms are included in our model

$$\begin{array}{ll}
\forall a, b.\ a \neq 0_F \ \wedge\ b \neq 0_F \implies a \times_F b \neq 0_F & \forall a, b.\ a \neq 0_F \implies a\ div_F\ b \neq 0_F \\
\forall a, b.\ a \neq b \implies a -_F b \neq 0_F & \forall a, b.\ a \neq -b \implies a +_F b \neq 0_F \\
\forall a, b.\ a \neq 0_F \implies a\ (**)_F\ b \neq 0_F & \forall a.\ a \neq 0_F \implies -_F a \neq 0_F
\end{array}$$

where F = field_mod_n_poly_$f(X)$. Literals of the extension field types are modeled in Jessie as vectors of polynomial coefficients. Therefore, logic functions to access and update the coefficient of a given power of some polynomial of type mod[mod[n] <X> / $p(X)$] are also included in the model, together with the usual two axioms for the theory of arrays.

$$\text{field\_mod\_}n\text{\_poly\_}f(x)\text{\_get\_coef} : \text{field\_mod\_}n\text{\_poly\_}f(x) \to \text{integer} \to \text{mod\_}n$$

$$\begin{aligned}\text{field\_mod\_}n\text{\_poly\_}f(x)\text{\_set\_coef} : \text{field\_mod\_}n\text{\_poly\_}f(x) \to \text{integer} \to \text{mod\_}n \\ \to \text{field\_mod\_}n\text{\_poly\_}f(x)\end{aligned}$$

The null polynomial is represented by field_$\langle\tau\rangle$_poly_$f(x)$_zero logical variable. An auxiliary axiom expresses that all of its coefficients are the zero element in τ.

Returning to the example introduced above, it can be seen by examining the type declaration of GF2C that the base type of an extension field can actually be an extension field itself. However, our modeling approach is exactly the same for this case, taking into consideration that the base type must be adjusted when defining the ring of polynomials over the base field.

$\mathsf{bits[n]}n \Rightarrow \mathsf{int}$	$\mathsf{mod}[\tau\ \text{<}X\text{>}\ /\ p(X)] \rightarrow \mathsf{vector}[n]\ \mathsf{of}\ \tau$
$\mathsf{mod}[n] \rightarrow \mathsf{int}$	$\mathsf{matrix}[1,n]\ \mathsf{of}\ \tau\ of\ \tau \rightarrow \mathsf{vector}[n]\ \mathsf{of}\ \tau$
$\mathsf{int} \rightarrow \mathsf{bits[n]}n$	$\mathsf{matrix}[n,1]\ \mathsf{of}\ \tau \rightarrow \mathsf{vector}[n]\ \mathsf{of}\ \tau$
$\tau \Rightarrow \mathsf{mod}[\tau\ \text{<}X\text{>}\ /\ p(X)]$	$\mathsf{vector}[n]\ \mathsf{of}\ \tau \rightarrow \mathsf{matrix}[1,n]\ \mathsf{of}\ \tau$
$\mathsf{vector}[n]\ \mathsf{of}\ \tau \rightarrow \mathsf{mod}[\tau\ \text{<}X\text{>}\ /\ p(X)]$	$\mathsf{vector}[n]\ \mathsf{of}\ \tau \rightarrow \mathsf{matrix}[n,1]\ \mathsf{of}\ \tau$

Figure 6.4: Casts (→) and coercions (⇒)

6.3.3 Structured types

As in C, CAO structured types aggregate a fixed number of fields, possibly of different types, into a single type. Typically, the struct type operations are access and update to struct fields, hence the Jessie model for the CAO structs is very similar to the vectors model. To access and update each field $\mathsf{field}_i : \tau_i$, the following two logic functions are declared:

$$\mathsf{struct\_}\langle\tau\rangle\mathsf{\_get\_field}_i \ :\ \mathsf{struct\_}\langle\tau\rangle \rightarrow \langle\tau_\mathsf{i}\rangle$$
$$\mathsf{struct\_}\langle\tau\rangle\mathsf{\_set\_field}_i \ :\ \mathsf{struct\_}\langle\tau\rangle \rightarrow \langle\tau_\mathsf{i}\rangle \rightarrow \mathsf{struct\_}\langle\tau\rangle$$

The behavior of these functions is axiomatized as expected, although it is slightly more verbose in order to deal with the fact that our index into the structure is now an identifier rather than an integer.

6.3.4 Casts and coercions

Type conversion operations in CAO can be explicit, in which case they are called *cast* operations, or implicit, called *coercion* operations. Figure 6.4 presents the allowed cast (→) and coercion (⇒) operations between CAO types. The translation of CAO programs into Jessie handles these conversions in the natural way by using appropriate logical functions. We present a few examples of the simpler conversions:

$$\begin{array}{rcl}
e :: \mathsf{mod}[n] & \Longrightarrow & \langle(\mathsf{int})\ \mathsf{e}\rangle \ = \ \mathsf{int\_of\_mod\_}n(\langle\mathsf{e}\rangle) \\
e :: \mathsf{int} & \Longrightarrow & \langle(\mathsf{mod[n]})\ \mathsf{e}\rangle \ = \ \mathsf{mod\_}n\mathsf{\_of\_int}(\langle\mathsf{e}\rangle) \\
e :: \mathsf{int} & \Longrightarrow & \langle(\mathsf{bits[n]n})\ \mathsf{e}\rangle \ = \ \mathsf{bits\_of\_int}(\langle\mathsf{e}\rangle) \\
e :: \tau & \Longrightarrow & \langle(\mathsf{mod[f(X)\ \text{<}X\text{>}\ /\ p(X)]})\ \mathsf{e}\rangle = \\
& & \quad \mathsf{field\_}\langle\tau\rangle\mathsf{\_poly\_}f(x)\mathsf{\_set\_coef}(\mathsf{field\_}\langle\tau\rangle\mathsf{\_poly\_}f(x)\mathsf{\_zero}, 0, \langle\mathsf{e}\rangle)
\end{array}$$

Conversions between matrices and column/row vectors are handled in the natural way by using get and set operations. Finally, we present the conversion between extension field types and vector types in more detail, since these are very useful CAO operators that permit commuting between the abstract algebraic view of a finite field, and its concrete representation in a cryptographic implementation. Indeed, one can construct an extension field value from a vector representation that contains the coefficients of the corresponding polynomial over the base field. We model this as

$$\begin{array}{l}\langle(\text{mod[f(X) <X> / p(X)]) e}\rangle \ = \\ \quad \textbf{let } x_1 \ = \text{field\_}\langle\tau\rangle\text{\_poly\_}f(x)\text{\_zero } \textbf{in } (\ \textbf{let } x_2 \ = \langle\text{e}\rangle \textbf{ in} \\ \quad \textbf{let } x_3 \ = \text{field\_}\langle\tau\rangle\text{\_poly\_}f(x)\text{\_set\_coef}(x_2, n-1, \text{vector\_}\langle\tau\rangle\text{\_get}(x_2, n-1)) \textbf{ in} \ ... \\ \quad \textbf{let } x_{n+2} \ = \text{field\_}\langle\tau\rangle\text{\_poly\_}f(x)\text{\_set\_coef}(x_{n+1}, 0, \text{vector\_}\langle\tau\rangle\text{\_get}(x_2, 0)) \textbf{ in } x_{n+2})\end{array}$$

The inverse conversion is also possible, and is modeled using a similar approach. This translation further justifies our modeling of extension field literals presented in the previous section.

6.3.5 Automatic safety proof obligations

Following the same approach adopted in tools such as Frama-c, the CAO to Jessie translation in our tool ensures that all statements in the input program that could potentially result in a safety violation originate the automatic generation of a verification condition that, if proven, guarantees the safe execution of the verified code. In chapter 7 we address the soundness of this approach in more detail, by showing that the validity of these safety conditions indeed implies that the program does not go wrong.

We have two classes of safety proof obligations: those related with memory safety, and those related with algebraic operations. Some of the proof obligations are automatically generated by the Jessie tool, while others are explicitly introduced in the generated Jessie code as assertions, during the translation process. We have encountered examples of these assertions in the models for exponentiation and division operations presented above. Table 6.1 presents the proof obligations that are generated to ensure the safety of memory access and algebraic operations. Proof obligations automatically generated by the Jessie plug-in are signaled in the table, corresponding to those that originate from the use of the Jessie integer type.

To support the automatic verification of safety proof obligations, our tool also

Type	Operation	Proof Obligation	Auto
int	e_1/e_2	$e_2 \neq 0$	×
	$e_1 ** e_2$	$e_2 \geq 0$	
mod[n]n	e_1/e_2	gcd(int_of_mod_n(e_2), n) = 1 ∧ int_of_mod_n(e_2) ≠ 0	×
	$e_1 ** e_2$	$e_2 \geq 0$	
mod[τ <X> / $p(X)$]	e_1 / e_2	$e_2 \neq 0$	
vector[n] of τ	$v[e]$	$0 \leq \langle \text{e} \rangle < n$	
	v |> i, v <| i	$0 \leq \langle \text{i} \rangle < n$	
	$v[i..j]$	$0 \leq \langle \text{i} \rangle < n \wedge 0 \leq \langle \text{j} \rangle < n \wedge \langle \text{i} \rangle < \langle \text{j} \rangle$	
matrix[n_1,n_2] of τ	$m[e_1, e_2]$	$0 \leq \langle \text{e}_1 \rangle < n_1 \wedge 0 \leq \langle \text{e}_2 \rangle < n_2$	
	$m[i..j, k..l]$	$0 \leq \langle \text{i} \rangle < n_1 \wedge 0 \leq \langle \text{j} \rangle < n_1 \wedge 0 \leq \langle \text{k} \rangle < n_2 \wedge 0 \leq \langle \text{l} \rangle < n_2 \wedge \langle \text{i} \rangle < \langle \text{j} \rangle \wedge \langle \text{k} \rangle < \langle \text{l} \rangle$	
bits[n]	$b[e]$	$0 \leq \langle \text{e} \rangle < n$	
	b |> i, b <| i	$0 \leq \langle \text{i} \rangle < n$	
	$b \gg i, b \ll i$	$0 \leq \langle \text{i} \rangle < n$	

Table 6.1: Safety proof obligations

enriches the translated Jessie code with lemmas that capture some number theoretic assumptions that are implicit in the type checking procedure. We believe that this approach is also useful in raising the programmer's awareness as to the necessity to ensure that these assumptions are true. Concretely, when an extension field is declared, our tool automatically generates lemmas that capture the necessary conditions for these declarations to be meaningful according to the CAO semantics. For extensions mod[mod[n] <X> / $p(X)$], our tool generates lemmas for the following two predicates:

is_prime(n)   ring_mod_n_is_irreducible(field_mod_n_poly_$f(x)$_generator)

When the base type for the extension is already an extension field, only the irreducibility lemma is generated.

Lemmas can be immediately used in proofs, so for instance the first lemma above can be used as an hypothesis in all proof obligations related to division operations in mod[n], requiring that the divisor is relative prime to the modulus. We emphasize, however, that the presence of lemmas also originates new proof obligations corresponding to the validation of the lemmas themselves.

6.4 Case studies

We addressed the formal verification and validation of the central component of the CAO implementation of the NaCl library: the crypto_box component. This component provides the most fundamental operation in a cryptographically protected network protocol, which is public-key authenticated encryption. In this section we present some of the results achieved in the verification of this component using CAOVerif.

6.4.1 Elliptic-curve scalar multiplication in NaCl

In this section, we present a case study extracted from the CAO implementation of the core component in the open-source NaCl cryptographic library. This component is responsible for carrying out the high-speed elliptic-curve computations required to perform a Diffie-Hellman secret key agreement protocol. At the high-level, given an elliptic curve point (p in the code, and typically a public key) and a scalar (n in the code, and typically a secret key), this component essentially calculates the result of repeatedly adding the given point to itself, where the number of additions is given by the integer value of the scalar. Here, addition should be understood as the group operation defined over the set of points of the particular elliptic curve implemented in NaCl.

The CAO source code for this component is presented in Appendix C and corresponds to a direct transcription of the NaCl specification. The functionality offered by this source code can be summarized as follows.

The entry point into the component is the crypto_scalarmult function, which takes as input two 32-byte arrays. This function then recovers the representation of the elliptic curve point using the unpack function, and also the secret key as a bitstring using the clampC function. Function curve25519 is then called to actually perform the elliptic curve computations. This function implements an exponentiation algorithm over a representation of the curve proposed by Montgomery [85]. The exponentiation algorithm in function curve25519 uses as sub-routines the actual curve addition and doubling (adding a point to itself) operations implemented by functions addMont and doubleMont, respectively. These functions operate over a representation of curve points that stores two coordinates x and z, which is captured by the structured type MontRep.

Before presenting our verification results for this case study we first present a small example of the output of our translation into the Jessie input language. This corresponds to function crypto_scalarmult.

```
vector_bits jc_crypto_scalarmult(vector_bits jc_n_input,
                                 vector_bits jc_p_input)
{
    var vector_bits jc_n = jc_n_input;
    var vector_bits jc_p = jc_p_input;
    var mod_25519 jc_pm = mod_25519_of_integer(
                              integer_of_bits(jc_unpack(jc_p)));
    var bits jc_nc = jc_clampC(jc_n);
    jc_a2 = mod_25519_of_integer(0);
    return jc_pack(bits_of_integer(integer_of_mod_25519(
                              jc_curve25519(jc_nc, jc_pm))))
}
```

Safety verification Passing the CAO code in the implementation of Appendix C to the verification tool without any annotations gives rise to 309 automatically generated safety proof obligations, most of them arising from accesses to vectors and bitstrings. Of these, only 4 proof obligations are not automatically proven by Alt-Ergo, all of them corresponding to function curve25519:

- One VC stating that index i in the bitstring access at line 49 is within bounds.
- Two VCs that aim to guarantee loop termination (these are inserted automatically by the Jessie back-end).
- One VC stating that the division in line 63 is safe: the tool can determine that the divisor is not zero because of the test condition in the if statement, but it cannot establish that the divisor is coprime to the modulus $2^{255} - 19$.

The loop annotations in lines 46-47 and the lemma establishing as an hypothesis that $2^{255} - 19$ is a prime number[4] in line 10 are enough to enable the tool to automatically discharge all proof obligations.

Functional correctness verification To illustrate how our tool can be used to address arbitrary verification goals we introduce a simple example aiming to establish the correctness of function clampC. This function is informally described in the NaCl specification as follows. "*ClampC maps* $(a_0, a_1, ..., a_{30}, a_{31})$ *to* $(a_0 - (a_0 \mod 8), a_1, ..., a_{30}, 64 + (a_{31} \mod 64))$. *In other words, ClampC clears bits* $(7, 0, \ldots, 0, 0, 128)$ *and sets bit*

[4]Of course this lemma appears as a non-verified proof obligation at the end of the verification run and one can only hope to verify it interactively.

$(0, 0 \ldots, 0, 0, 64)$." Here, the bits to be cleared and bits to be set are specified by the one-bits when the provided values are seen as 8-bit words. The postcondition for clampC in lines 79-83 captures this specification.

In order to verify that the clampC function indeed satisfies this specification, we first needed to annotate function unpack with a postcondition (lines 66-68), as this is used by clampC to compute its final result. We also added a set of assertions to guide automatic provers into intermediate verification results that allow them to automatically discharge parts of the postcondition for clampC.

With the annotations included in the code (available in Appendix C) the CAO verification tool is able to discharge all but 1 proof obligation automatically (at least within reasonable time): the postcondition for function unpack. This is essentially due to the large number of nested logic function applications resulting from the translation of the concatenation operations: concatenation is translated in Jessie as a *blit* operation, hence for 31 concatenations we will have 31 nested *blit* operations. We conclude this section with a short description of how we validated this proof obligation using the Coq proof assistant.

The postcondition expresses that, for $0 \leq i < 31$ and $0 \leq j < 7$, the result of accessing the j-th bit in the i-th bitstring in the input vector n is the same as that of accessing the $8i + j$-th position in the concatenated bitstring result returned by the function. A simple proof strategy is to exhaustively traverse all the relevant values of i and j and establishing that equality indeed holds. We adopted this strategy, but rather than manually expanding all 256 proof iterations, we developed a simple Coq tactic that implements it based on the following simple lemma.

$$\forall a, b.\, b \leq a \vee P(a) \wedge (\forall i.\, a + 1 \leq i < b \implies P(i)) \implies (\forall i.\, a \leq i < b \implies P(i))$$

Here, P is instantiated with the property we want to prove (in[i][j]=result[i*8+j]), parameterized by the values of i; and a and b are instantiated with the lower and upper bounds of i, respectively. Note that the proof strategy adopted here is multi-tiered in the sense that first we try to automatically discharge all the proof obligations and then use Coq to prove the remaining verification conditions.

6.4.2 Minimizing exposure to side channel attacks in NaCl core

One of the features of the NaCl cryptographic library is that its implementation enforces strict security policies (already introduced in Chapter 5) that aim to minimize exposure to known side channel attacks. Recall that in Chapter 5 we also presented a case study related with the formal verification of these policies in the C implementation of the NaCl library. In this section we show how CAOVerif can be used to formal verify such policies in the CAO implementation of NaCl. To illustrate this point we will use as an example the implementation of the cryptographic function crypto_verify implemented in CAO, defined below.

```
def crypto_verify(x :  vector[16] of unsigned bits[8],
                  y :  vector[16] of unsigned bits[8])  :  unsigned bits[32] {
  def differentbits : unsigned bits[32];
  seq i := 0 to 15 {differentbits[0..7] := differentbits[0..7] | (x[i] ^ y[i]);}

  differentbits := (unsigned bits[32])(differentbits - 1) >> 8;
  return (unsigned bits[32])(differentbits[0] - 1);
}
```

The verification technique we adopt is a direct adaptation of the results presented in Chapter 5 where we explored the use of the deductive verification functionality of Frama-c to validate information flow properties of C programs.

The annotated CAO code that can be fed to CAOVerif in order to conduct this proof is shown in Figure 6.1. Despite the considerable effort required to produce adequate annotations, the verification itself is fully automatic for this simple example. We highlight that the annotation process can be automated.

6.5 Summary

We have presented CAOVerif, a deductive verification tool for CAO. It relies on the Jessie plug-in of the Frama-c framework as a back-end, and so we translate CAO annotated programs into the Jessie input language. CAO has a mathematically rich type system, designed to facilitate the implementation of cryptographic primitives, and in turn, Jessie's type system only includes booleans, arbitrary precision integers, reals and the unit type. Therefore, the translation also includes a first-order logic model of these mathematical objects that have specific interest for cryptography. Beyond that, our tool automatically generates the verification conditions related with the safety of CAO programs. We have illustrated the operation of our tool in real-world examples.

```
/*@ axiomatic list{
    logic list;
    logic list cons(n : int, s : list);
    } */

/*@ ghost def memory_x1, memory_x2 : int; */
/*@ ghost def memory_y1, memory_y2 :int; */
/*@ ghost def memory_db1, memory_db2 : int; */
/*@ axiomatic logic_mem { logic list f(i : int); }
 */

/*@ ensures f(result) == cons(offset,f(at(i,Old))) */
def append(i : int, offset : int) : int { return i; }

/*@ requires (memory_x1 == memory_x2) && (memory_y1 == memory_y2) &&
            (memory_db1 == memory_db2)
 */
def crypto_verify_selfcomp(x1,x2 : vector[16] of unsigned bits[8],
                           y1,y2 : vector[16] of unsigned bits[8])
: unsigned bits[32], unsigned bits[32]
{

  def differentbits1, differentbits2 : unsigned bits[32];

  seq i := 0 to 15 {
      differentbits1[0..7] := differentbits1[0..7] | (x1[i] ^ y1[i]);
      /*@ghost seq j := 0 to 7 {
           memory_db1 := append(memory_db1,j);
           }
          memory_x1 := append(memory_x1,i);
          memory_y1 := append(memory_y1,i);
       */
  }
  differentbits1 := (unsigned bits[32])(differentbits1   1) >> 8;
  differentbits1 := (unsigned bits[32])(differentbits1[0]   1);
  /*@ ghost memory_db1 := append(memory_db1,0); */

  seq i := 0 to 15 {
      differentbits2[0..7] := differentbits2[0..7] | (x2[i] ^ y2[i]);
        /*@ghost seq j := 0 to 7 {
         memory_db2 := append(memory_db2,j);
         }
         memory_x2 := append(memory_x2,i);
         memory_y2 := append(memory_y2,i);
       */
  }
  differentbits2 := (unsigned bits[32])(differentbits2   1) >> 8;
  differentbits2 := (unsigned bits[32])(differentbits2[0]   1);
  /*@ ghost memory_db2 := append(memory_db2,0); */

  /*@ assert (f(memory_x1) == f(memory_x2)) &&
           (f(memory_y1) == f(memory_y2)) &&
           (f(memory_db1) == f(memory_db2)) */
  return differentbits1, differentbits2;
}
```

Listing 6.1: Annotated CAO implementation of crypto_verify function

Chapter 7

Establishing the soundness of CAOVerif

In the previous chapter we introduced CAOVerif, a deductive verification tool for CAO. This tool relies on the Jessie plug-in of the Frama-c framework. CAOVerif translates each CAO annotated program into Jessie input language together with a rich formalisation of the CAO type-system in first-order logic.

At this point, it makes sense to ask what properties the translation of CAO programs into Jessie (and the accompanying models) should enjoy. One goal we discussed in the previous chapter was to enable as many assertions as possible to be proved automatically; more precisely, the verification conditions produced by Jessie, and exported to some external theorem prover, should as much as possible be discharged automatically. For this, the models must describe the operations of each data type as completely as possible, in such a way that one can reason about the widest possible program properties, yet striking a compromise so as not to overwhelm the back-end provers.

Soundness is of course an essential property: the Jessie translation should not allow proving assertions about CAO programs that are not valid according to the language semantics. CAOVerif relies on the Jessie plug-in which in turn relies itself on the Why platform. Thus the soundness of CAOVerif depends of the soundness of the subsequent tools of this chain. In this work we will assume the correctness of the VCGen for the Jessie language to establish the soundness of CAOVerif. In other words that is, we will assume that the validity of the proof obligations generated by the Why tool imply the correctness of the original Jessie program. This assumption is supported by the work that is being developed by the Frama-c developers in the certification of the chain of the Jessie plug-in of the Frama-c framework [52, 53].

Another desirable property would be *Completeness*, i.e. showing that everything correct program can be so proven using the Jessie model generated by CAOVerif. However, hoping for a completeness proof would be too ambitious, as our approach is based on a compromise between expressiveness and automation in which some of the models we generate are not complete (e.g. extension fields and matrices). A less ambitious goal would be to formalize and reason about a notion of adequacy of our VCGen tool. Intuitively, this would state that the translation is *useful* in the sense that, i.e. it does not map all programs into *dummy* Jessie programs that can never be proven correct. To argue in this direction one could rely on the fact that our translation is total and injective, i.e. we can translate all CAO programs into Jessie programs, and then translate it back to CAO (ignoring the safety assertions).

In the remainder of this chapter we will formally reason about the validity of this approach by formally proving that our strategy for constructing CAOVerif using the Jessie plug-in as a back-end is sound.

7.1 Proof strategy

Literature contains various examples of soundness proofs for VCGens [55, 18, 92]. According to Homeier et. al [55], the soundness of the VCGen for *While^C* (the language introduced in Chapter 2) can be established by proving the following theorem.

Theorem 1. *Let $\{\!|\varphi|\!\}\ C\ \{\!|\psi|\!\}$ be a safety-sensitive Hoare triple in While^C.*

$$\forall s \in \mathsf{State}.\ [\![\mathsf{VCG}_{\mathsf{while}^C}(\{\!|\varphi|\!\}\ C\ \{\!|\psi|\!\})]\!]_{\mathcal{M}}(s) = \mathbf{true} \implies [\![\{\!|\varphi|\!\}\ C\ \{\!|\psi|\!\}]\!]_{\mathcal{M}}(s) = \mathbf{true}$$

A direct proof strategy would rely on the formal definition of the VCGen and the axiomatic semantics of the language to show that validity of the generated verification conditions implies correctness of the program. However, given our implementation strategy, a more natural approach is to modularize the proof by assuming the soundness of the Jessie VCGen, and arguing that our translation ensures that the soundness CAOVerif as a VCGen is implied of the soundness of the Jessie VCGen.

We refrain from presenting our soundness proof for the full versions of CAO and Jessie. Instead we give a proof of concept which highlights the most important aspects of this proof by using the abstractions of these languages we introduced in Chapter 2.

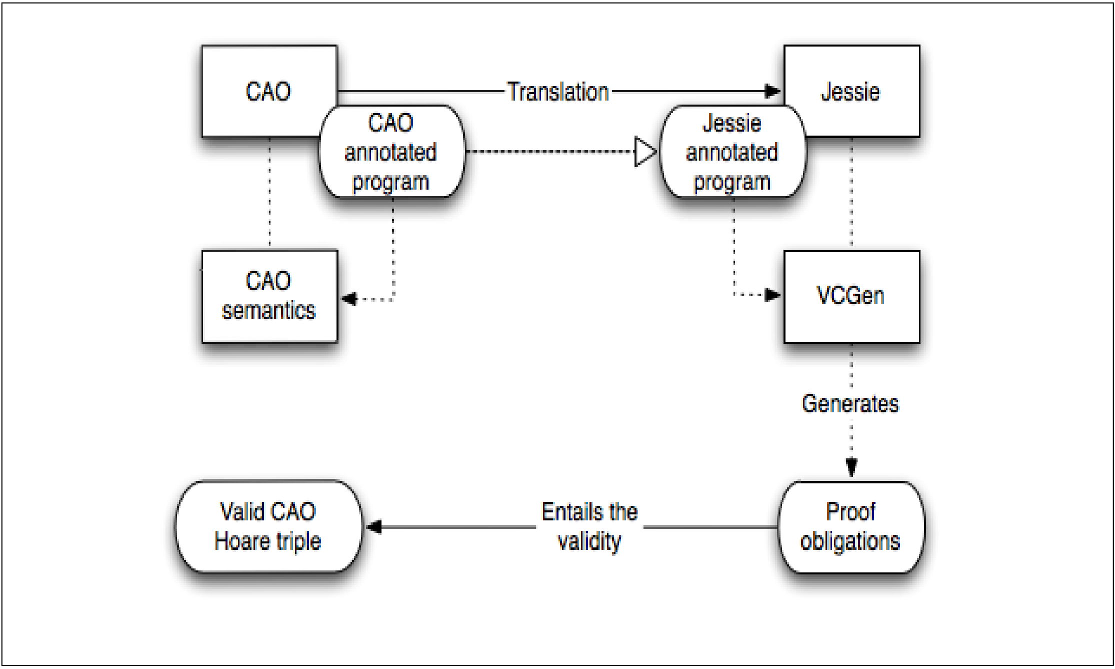

Figure 7.1: Proof strategy

To this end, we now present a formal definition of a translation from *While*^{C} to *While**, that corresponds to an abstraction of the real translation that is done in CAOVerif.

Translation (*While*^{C} **to** *While**) In CAOVerif we translate CAO annotated programs to Jessie annotated programs. So we define a translation from *While*^{C} annotated programs to *While** annotated programs given by $\langle\cdot\rangle$ and inductively defined by the rules presented in Figure 7.2. The predicate safe^* is responsible for generating the safety conditions related with error-free *While*^{C} expressions. This predicate abstracts what is done in the translation from CAO to Jessie, where for each program construction that could potentially result in a safety violation, the translation generates a similar assertion.

An important preliminary result is stated in Lemma 25. Intuitively, this establishes that the validity of any assertion resulting from our translation into *While** implies the validity of the original assumption under the semantics of *While*^{C}.

Lemma 25. *Let φ be an assertion in While^C and $\langle\varphi\rangle$ its translation in While*.*

$$\models_{\mathcal{T}} \langle\varphi\rangle \quad\Longrightarrow\quad \forall s \in \mathsf{State}.\ [\![\varphi]\!]_{\mathcal{M}}(s) = \boldsymbol{true}$$

Proof. Intuitively, one has to prove that all the sentences derivable in the theory $\mathcal{T}$, the

$$\langle \{\!|\varphi|\!\}\ C\ \{\!|\psi|\!\}\rangle = \{\langle\varphi\rangle\}\ \langle C\rangle\ \{\langle\psi\rangle\}$$

$$\begin{array}{llll}
\langle \mathbf{n}\rangle = \mathbf{n} & \langle \mathrm{x}\rangle = \mathrm{x} & \langle e^i_1\ \mathsf{op}\ e^i_2\rangle = \langle e^i_1\rangle\ \mathsf{op}\ \langle e^i_2\rangle & \mathsf{op} \in \{+,-,*,<,>,/\} \\
\langle \mathbf{true}\rangle = \mathbf{true} & \langle \mathbf{false}\rangle = \mathbf{false} & \langle e^b_1\ \mathsf{op}\ e^b_1\rangle = \langle e^b_1\rangle\ \mathsf{op}\ \langle e^b_2\rangle & \mathsf{op} \in \{!=,==\} \\
\langle \mathsf{a}[e]\rangle = \mathsf{acc}(\mathsf{a},\langle e\rangle) & & &
\end{array}$$

$$\begin{array}{rcl}
\langle \mathbf{skip}\rangle &=& \mathbf{skip} \\
\langle \mathtt{x} := e\rangle &=& \mathbf{assert}\ \mathsf{safe}^*(\mathsf{e});\ \mathtt{x} := \langle e\rangle \\
\langle \mathtt{a}[e_1] := e_2\rangle &=& \mathbf{assert}\ \mathsf{safe}^*(\mathsf{a}[\mathsf{e}_1]) \wedge \mathsf{safe}^*(\mathsf{e}_2);\ \mathsf{a} := \mathrm{set}(\mathsf{a},\langle e_1\rangle,\langle e_2\rangle) \\
\langle \mathbf{if}\ (e)\ \{C_1\}\ \mathbf{else}\ \{C_2\}\rangle &=& \mathbf{assert}\ \mathsf{safe}^*(\mathsf{e});\ \mathbf{if}\ (\langle e\rangle)\ \{\langle C_1\rangle\}\ \mathbf{else}\ \{\langle C_2\rangle\} \\
\langle \mathbf{while}\ \{\theta\}\ (e)\ \{C\}\rangle &=& \mathbf{while}\ \{\langle\theta\rangle \wedge \mathsf{safe}^*(\mathsf{e})\}\ (\langle e\rangle)\ \{\langle C\rangle\} \\
\langle C_1; C_2\rangle &=& \langle C_1\rangle;\ \langle C_2\rangle \\
\langle \mathbf{assert}\ \phi\rangle &=& \mathbf{assert}\ \langle\phi\rangle
\end{array}$$

$$\begin{array}{ccc}
\langle \mathsf{upd}(\mathsf{a},e_1,e_2)\rangle = \mathrm{set}(\mathsf{a},\langle e_1\rangle,\langle e_2\rangle) & \langle e_1\ \mathsf{op}\ e_2\rangle = \langle e_1\rangle\ \mathsf{op}\ \langle e_2\rangle & \mathsf{op} \in \{==,!=,<,>\} \\
\langle \psi \wedge \phi\rangle = \langle\psi\rangle \wedge \langle\phi\rangle & \langle \psi \vee \phi\rangle = \langle\psi\rangle \vee \langle\psi\rangle & \langle \psi \rightarrow \phi\rangle = \langle\psi\rangle \rightarrow \langle\phi\rangle \\
\langle \psi \leftrightarrow \phi\rangle = \langle\psi\rangle \leftrightarrow \langle\phi\rangle & \langle \forall x.\ \psi\rangle = \forall x.\langle\psi\rangle & \langle \exists x.\ \psi\rangle(s) = \exists x.\langle\psi\rangle
\end{array}$$

$$\begin{array}{rll}
\mathsf{safe}^*(\mathbf{n}) = \mathbf{true} & \mathsf{safe}^*(\mathsf{e}^{\mathsf{i}}_1\ op_{\mathsf{i}}\ \mathsf{e}^{\mathsf{i}}_2) = \mathsf{safe}^*(\mathsf{e}^{\mathsf{i}}_1) \wedge \mathsf{safe}^*(\mathsf{e}^{\mathsf{i}}_2) & op_i \in \{+,-,*,<,>\} \\
\mathsf{safe}^*(\mathrm{x}) = \mathbf{true} & \mathsf{safe}^*(\mathsf{e}^{\mathsf{b}}_1\ op_{\mathsf{b}}\ \mathsf{e}^{\mathsf{b}}_1) = \mathsf{safe}^*(\mathsf{e}^{\mathsf{b}}_1) \wedge \mathsf{safe}^*(\mathsf{e}^{\mathsf{b}}_2) & op_b \in \{!=,==\} \\
\mathsf{safe}^*(\mathbf{true}) = \mathbf{true} & \mathsf{safe}^*(\mathsf{a}[\mathsf{e}]) = \mathsf{safe}^*(\mathsf{e}) \wedge 0 \le \langle e\rangle < \mathsf{len}(\mathsf{a}); & \\
\mathsf{safe}^*(\mathbf{false}) = \mathbf{true} & \mathsf{safe}^*(\mathsf{e}^{\mathsf{i}}_1/\mathsf{e}^{\mathsf{i}}_2) = \mathsf{safe}^*(\mathsf{e}^{\mathsf{i}}_1) \wedge \mathsf{safe}^*(\mathsf{e}^{\mathsf{i}}_2) \wedge \langle e^i_2\rangle != 0; &
\end{array}$$

Figure 7.2: Translation from *While*^{C} to *While**

theory considered in *While**, can be validated by the model $\mathcal{M}$ that interprets expressions/operations according to the semantics of *While*^{C}. Observe that in both cases the difference lies on the arrays data type whose operations are axiomatized in *While** and defined in *While*^{C}. When a theory is defined by a set of axioms, then a sentence is valid in the theory only if it is derivable from the set of axioms.

By induction on the translation rules, one can deduce that any derivation tree that establishes the validity of a translated *While** assertion from the axioms in the generated theory, can be mapped onto an equally valid assertion in the *While*^{C} world. Using the translation rules (depicted in Figure 7.2), we have that,

$$\langle \mathsf{upd}(\mathsf{a}, e_1, e_2)\rangle = \mathrm{set}(\mathsf{a}, \langle e_1\rangle, \langle e_2\rangle)$$

so the axioms which constrain the array operations in *While** can be mapped back to *While*^{C} as:

$$\begin{aligned}
&\langle \mathsf{upd}(\mathsf{a}, e_1, e_2)[e_1] = e_2\rangle \equiv \ \mathrm{get}(\mathrm{set}(\mathsf{a}, \langle e_1\rangle, \langle e_2\rangle), \langle e_2\rangle) = \langle e_2\rangle \\
&e_1 \neq e_3 \rightarrow \mathsf{upd}(\mathsf{a}, e_1, e_2)[e_3] = \mathsf{a}[e_3] \equiv \ \langle e_1\rangle \neq \langle e_3\rangle \rightarrow \\
&\qquad\qquad \mathrm{get}(\mathrm{set}(\mathsf{a}, \langle e_1\rangle, \langle e_2\rangle), \langle e_3\rangle) = \mathrm{get}(\mathsf{a}, \langle e_3\rangle)
\end{aligned}$$

Therefore, to complete the proof, one simply needs to establish that these statements are valid under the *While*^{C} semantics. By definition,

$$\begin{aligned}
&[\![\mathsf{upd}(\mathsf{a}, e_1, e_2)[e_1] = e_2]\!]_{\mathcal{M}}(s) = \textbf{true} \equiv [\![\mathsf{upd}(\mathsf{a}, e_1, e_2)[e_1]]\!]_{\mathcal{M}}(s) = [\![e_2]\!]_{\mathcal{M}}(s) \equiv \\
&\qquad \mathsf{upd}(s(\mathsf{a}), s(e_1), s(e_2))(s(e_1)) = s(e_2) \\
&\text{and} \\
&[\![\mathsf{upd}(\mathsf{a}, e_1, e_2)[e_3] = \mathsf{a}[e_3]]\!]_{\mathcal{M}}(s) = \textbf{true} \equiv [\![\mathsf{upd}(\mathsf{a}, e_1, e_2)[e_3]]\!]_{\mathcal{M}} = [\![\mathsf{a}[e_3]]\!]_{\mathcal{M}} \equiv \\
&\qquad \mathsf{upd}(s(\mathsf{a}), s(e_1), s(e_2))(s(e_3)) = s(\mathsf{a})(s(e_3))
\end{aligned}$$

and using the definition of upd both equalities immediately hold.

□

Safety Safety assertions are introduced in the translation from *While*^{C} to *While** through the predicate safe$^*(\cdot)$. This predicate is defined for *While*^{C} expressions when the evaluation of some of the underlying expressions results in an **error** value. So, a natural result that is expected to hold is that the validity of safe*(e), for some *While*^{C} expression, implies that e does not evaluate to an **error** value. The result expressed by the following lemma, together with the result of Lemma 2 (from Chapter 2), entails the latter one.

Lemma 26. *Let e be an expression in While*^{C}.

$$\models_{\mathcal{T}} \mathsf{safe}^*(\mathsf{e}) \implies \forall s \in \mathsf{State}.\ [\![\mathsf{safe}(\mathsf{e})]\!]_{\mathcal{M}}(s) = \textbf{true}$$

Proof. By induction on the structure of e and using Lemma 25. We remark that a very important notion captured by this lemma is that for some expression e, we have $\langle \mathsf{safe(e)} \rangle = \mathsf{safe}^*(\mathsf{e})$. The less intuitive cases are when $e \equiv \mathsf{a}[e_1]$ and $e \equiv e_1/e_2$, and for those cases the proof is presented next.

- For $e \equiv \mathsf{a}[e_1]$, we have that $\mathsf{safe}^*(\mathsf{a[e_1]}) = \mathsf{safe}^*(\mathsf{e_1}) \wedge 0 \leq \langle e_1 \rangle < \mathsf{len(a)}$. By definition $\mathsf{safe(a[e_1])} = \mathsf{safe(e_1)} \wedge 0 \leq e_1 < \mathsf{len(a)}$, so we need to prove that $[\![\mathsf{safe(e_1)}]\!]_{\mathcal{M}}(s) = \mathbf{true}$ and $[\![0 \leq e_1 < \mathsf{len(a)}]\!]_{\mathcal{M}}(s) = \mathbf{true}$, for all state s. By induction hypothesis we can assume that $\models_{\mathcal{T}} \mathsf{safe}^*(\mathsf{e_1})$ and conclude that $[\![\mathsf{safe(e_1)}]\!]_{\mathcal{M}}(s) = \mathbf{true}$, for all state s, and since $0 \leq \langle e_1 \rangle < \mathsf{len(a)} \equiv \langle 0 \leq e_1 < \mathsf{len(a)} \rangle$, by Lemma 25 we conclude that $[\![0 \leq e_1 < \mathsf{len(a)}]\!]_{\mathcal{M}}(s) = \mathbf{true}$.
- For $e \equiv e_1/e_2$, we have that $\mathsf{safe}^*(\mathsf{e_1/e_2}) = \mathsf{safe}^*(\mathsf{e_1}) \wedge \mathsf{safe}^*(\mathsf{e_2}) \wedge \langle e_2 \rangle \neq 0$. By definition, $\mathsf{safe(e_1/e_2)} = \mathsf{safe(e_1)} \wedge \mathsf{safe(e_2)} \wedge e_2 \neq 0$.

  By induction hypothesis we can assume that $\models_{\mathcal{T}} \mathsf{safe}^*(\mathsf{e_1})$ and $\models_{\mathcal{T}} \mathsf{safe}^*(\mathsf{e_2})$ and conclude that $[\![\mathsf{safe(e_1)}]\!]_{\mathcal{M}}(s) = \mathbf{true}$ and $[\![\mathsf{safe(e_2)}]\!]_{\mathcal{M}}(s) = \mathbf{true}$, for all state s. Now, observing that $\langle e_2 \rangle \neq 0 \equiv \langle e_2 \neq 0 \rangle$, by Lemma 25 we can conclude that $[\![e_2 \neq 0]\!]_{\mathcal{M}}(s) = \mathbf{true}$, for all state s.

□

Soundness We now use the previous result to prove that when a program is translated according to the rules depicted in Figure 7.2, a derivation tree establishing the correctness of the translated program under the *While** axiomatic semantics implies the existence of a valid derivation tree establishing the correctness for the original program.

Lemma 27. *Let* $\{\!|\varphi|\!\}\ C\ \{\!|\psi|\!\}$ *be a safety-sensitive Hoare triple in* While^C *and* $\{\langle\varphi\rangle\}\langle C\rangle$ $\{\langle\psi\rangle\}$ *its translation in* While^*. *Let also* $\vdash_{\mathit{While}^C} \{\!|\varphi|\!\}\ C\ \{\!|\psi|\!\}$ *denote that* $\{\!|\varphi|\!\}\ C\ \{\!|\psi|\!\}$ *is a Hoare triple derivable using the axioms and rules from the inference system of Figure 2.3 from Chapter 2. Then, we have*

$$\vdash_{\mathcal{T}} \{\langle\varphi\rangle\}\ \langle C\rangle\ \{\langle\psi\rangle\} \implies \vdash_{\mathit{While}^C} \{\!|\varphi|\!\}\ C\ \{\!|\psi|\!\}$$

Proof. By induction on the structure of the While^C programs. We just present the proof for $C \equiv \mathsf{a}[e_1] := e_2$, $C \equiv \mathbf{if}\ (e)\ \{C_1\}\ \mathbf{else}\ \{C_2\}$ and $C \equiv \mathbf{while}\ \{\theta\}\ (e)\ \{C\}$, since these are the more interesting cases.

The proof strategy is as follows: assuming the existence of a single derivation tree for each $\{\langle\varphi\rangle\}\ \langle C\rangle\ \{\langle\psi\rangle\}$, where all of the side-conditions are valid, we need to prove that there is a derivation tree for the corresponding $\{\!|\varphi|\!\}\ C\ \{\!|\psi|\!\}$, where all of the side-conditions are valid too. Using the axioms and rules of the axiomatic semantics of *While*^{C} and *While**, one can derive the proof trees shown in Figure 7.3[1].

- $\{\!|\varphi|\!\}\ \mathsf{a}[e_1] := e_2\ \{\!|\psi|\!\}$

  The only thing that is left to prove in the derivation tree of $\{\!|\varphi|\!\}\ \mathsf{a}[e_1] := e_2\ \{\!|\psi|\!\}$ is that its side-condition A_3 is valid. Since we have assumed that A_1 and A_2 are valid, by transitivity of the implication follows that, $\models_{\mathcal{T}} (\langle\varphi\rangle \rightarrow \mathsf{safe}^*(\mathsf{a}[\mathsf{e}_1]) \wedge \mathsf{safe}^*(\mathsf{e}_2)) \wedge (\langle\varphi\rangle \rightarrow \langle\psi\rangle[\mathrm{set}(\mathsf{a}, \langle e_1\rangle, \langle e_2\rangle)/\mathsf{a}])$. By Lemma 25, immediately follows that $[\![(\varphi \rightarrow \mathsf{safe}(\mathsf{a}[\mathsf{e}_1])) \wedge (\varphi \rightarrow \mathsf{safe}(\mathsf{e}_2)) \wedge (\varphi \rightarrow \psi[\mathsf{upd}(\mathsf{a}, \langle e_1\rangle, \langle e_2\rangle)/\mathsf{a}])]\!](s) =$ **true**, for any state s.

- $\{\!|\varphi|\!\}$ **while** $\{\theta\}\ (e)\ \{C\}\ \{\!|\psi|\!\}$

  As it is shown in Figure 7.3, to prove that $\vdash_{While^C} \{\!|\varphi|\!\}$ **while** $\{\theta\}\ (e)\ \{C\}\ \{\!|\psi|\!\}$, we need to prove that

  1. $\vdash_{While^C} \{\!|\theta \wedge \mathsf{safe}(\mathsf{e}) \wedge e|\!\}\ C\ \{\!|\theta \wedge \mathsf{safe}(\mathsf{e})|\!\}$ and
  2. $[\![(\varphi \rightarrow \theta \wedge \mathsf{safe}(\mathsf{e})) \wedge (\theta \wedge \mathsf{safe}(\mathsf{e}) \wedge \neg e \rightarrow \psi)]\!]_{\mathcal{M}}(s) =$ **true**, for all state s.

  By induction hypothesis and using Lemma 26 we can conclude (1), since we assume the existence of a derivation tree in *While** for $\{\langle\theta\rangle \wedge \mathsf{safe}^*(\mathsf{e}) \wedge \langle e\rangle\}\ \langle C\rangle\ \{\langle\theta\rangle \wedge \mathsf{safe}^*(\mathsf{e})\}$.

  Now using the fact that $\models_{\mathcal{T}} \langle\varphi\rangle \rightarrow \mathsf{safe}^*(\mathsf{e}) \wedge \langle\theta\rangle$ and $\models_{\mathcal{T}} (\langle\theta\rangle \wedge \mathsf{safe}^*(\mathsf{e}) \wedge \neg\langle e\rangle \rightarrow \langle\psi\rangle)$, by Lemmas 25 and 26 we can easily conclude (2).

- $\{\!|\varphi|\!\}$ **if** $(e)\ \{C_1\}$ **else** $\{C_2\}\ \{\!|\psi|\!\}$

  To prove that $\vdash_{While^C} \{\!|\varphi|\!\}$ **if** $(e)\ \{C_1\}$ **else** $\{C_2\}\ \{\!|\psi|\!\}$ one needs to prove that,

  1. $\vdash_{While^C} \{\!|\varphi \wedge e|\!\}\ C_1\ \{\!|\psi|\!\}$
  2. $\vdash_{While^C} \{\!|\varphi \wedge \neg e|\!\}\ C_2\ \{\!|\psi|\!\}$ and
  3. $\varphi \rightarrow \mathsf{safe}(\mathsf{e})$.

     By induction hypothesis

$$\vdash_{\mathcal{T}} \{\rho \wedge \langle e\rangle\}\ \langle C_1\rangle\ \{\langle\psi\rangle\} \qquad\qquad \vdash_{\mathcal{T}} \{\rho \wedge \neg\langle e\rangle\}\ \langle C_2\rangle\ \{\langle\psi\rangle\}$$

[1]Observe that these results are valid for any intermediate assertion ρ, that may be considered in the construction of the derivation tree of each translated Hoare triple.

and since $\models_{\mathcal{T}} \langle\varphi\rangle \rightarrow \rho$ (by assertion A_1), immediately follows that[2]

$$\vdash_{\mathcal{T}} \{\langle\varphi\rangle \wedge \langle e\rangle\}\ \langle C_1\rangle\ \{\langle\psi\rangle\} \qquad\qquad \vdash_{\mathcal{T}} \{\langle\varphi\rangle \wedge \neg\langle e\rangle\}\ \langle C_2\rangle\ \{\langle\psi\rangle\}$$

and we can easily conclude (1) and (2). Now using Lemma 25 and assertion A_2 we can easily conclude (3).

$\square$

Finally, using the previous results we can establish the soundness of the VCGen for *While*^{C}. Recall that we rely on the fact that the VCGen of Jessie/*While** is correct, so, for each (translated) Hoare triple whose verification conditions are proved to be valid, there exists a derivation tree where all the side-conditions are also valid.

Theorem 2. *(VCGen soundness) Let* $\{\!|\varphi|\!\}\ C\ \{\!|\psi|\!\}$ *be a Hoare triple in While*^{C} *and* $\{\langle\varphi\rangle\}\{\langle C\rangle\}$ $\{\langle\psi\rangle\}$ *its translation in While**.

$$\models_{\mathcal{T}} \mathsf{VCG}_{\mathsf{while}^*}(\{\langle\varphi\rangle\}\ \langle C\rangle\ \{\langle\psi\rangle\}) \implies \forall s \in \mathsf{State}.[\![\{\!|\varphi|\!\}\ C\ \{\!|\psi|\!\}]\!]_{\mathcal{M}}(s) = \boldsymbol{true}$$

Proof. Using the Lemma 27 above defined and Lemma 3 and Proposition 5, from Chapter 2. $\square$

7.2 Summary

We have presented a proof of concept of how one can prove the soundness of CAO-Verif. Our work distinguishes itself from the other works such as [55, 18, 92, 52], since the tool responsible for the generation of the verification conditions is not built in the target language. The proof obligations are generated for a program that results from the translation of the original program (CAO) to an intermediate language (Jessie). In fact, we prove that the correctness of the translated program entails the correctness of the original program. Of course our result would be ill-founded if we did not rely on the correctness of the Jessie VCGen. However, as stated, this result is being independently established by the Frama-c team [52].

A very important result achieved is that we can guarantee that the CAO programs safely execute, if all the safety proof obligations (inserted during the translation) are proved correct.

[2]Note that this proof can be easily done by simple induction on the structure of *While** programs.

$$(\textsc{seq})\ \dfrac{(\textsc{assert})\ \dfrac{}{\{\langle\varphi\rangle\}\ \textbf{assert}\ \mathsf{safe}^*(\mathsf{a}[\mathsf{e}_1]) \wedge \mathsf{safe}^*(\mathsf{e}_2)\ \{\rho\}}\ A_1 \qquad (\textsc{assign})\ \dfrac{}{\{\rho\}\ \mathsf{a} = \mathrm{set}(\mathsf{a}, \langle e_1\rangle, \langle e_2\rangle)\ \{\langle\psi\rangle\}}\ A_2}{\{\langle\varphi\rangle\}\ \textbf{assert}\ \mathsf{safe}^*(\mathsf{a}[\mathsf{e}_1]) \wedge \mathsf{safe}^*(\mathsf{e}_2);\ \mathsf{a} = \mathrm{set}(\mathsf{a}, \langle e_1\rangle, \langle e_2\rangle)\ \{\langle\psi\rangle\}}$$

$$(\textsc{assign})\ \dfrac{}{\{\!|\varphi|\!\}\ \mathsf{a}[e_1] := e_2\ \{\!|\psi|\!\}}\ A_3$$

where $A_1 : \langle\varphi\rangle \to \mathsf{safe}^*(\mathsf{a}[\mathsf{e}_1]) \wedge \mathsf{safe}^*(\mathsf{e}_2) \wedge \rho$, $A_2 : \rho \to \langle\psi\rangle[\mathrm{set}(\mathsf{a}, \langle e_1\rangle, \langle e_2\rangle)/\mathsf{a}]$ and $A_3 : \varphi \to \mathsf{safe}(\mathsf{a}[\mathsf{e}_1]) \wedge \varphi \to \mathsf{safe}(\mathsf{e}_2) \wedge \varphi \to \psi[\mathsf{upd}(\mathsf{a}, \langle e_1\rangle, \langle e_2\rangle)/\mathsf{a}]$.

$$(\textsc{while})\ \dfrac{\{\langle\theta\rangle \wedge \mathsf{safe}^*(\mathsf{e}) \wedge \langle e\rangle\}\ \langle C\rangle\ \{\langle\theta\rangle \wedge \mathsf{safe}^*(\mathsf{e})\}}{\{\langle\varphi\rangle\}\ \textbf{while}\ \{\langle\theta\rangle \wedge \mathsf{safe}^*(\mathsf{e})\}\ (\langle e\rangle)\ \{\langle C\rangle\}\ \{\langle\psi\rangle\}}\ A_1 \qquad (\textsc{while})\ \dfrac{\{\!|\theta \wedge \mathsf{safe}(\mathsf{e}) \wedge e|\!\}\ C\ \{\!|\theta \wedge \mathsf{safe}(\mathsf{e})|\!\}}{\{\!|\varphi|\!\}\ \textbf{while}\ \{\theta\}\ (e)\ \{C\}\ \{\!|\psi|\!\}}\ A_2$$

where $A_1 : (\langle\varphi\rangle \to \langle\theta\rangle \wedge \mathsf{safe}^*(\mathsf{e})) \wedge\ (\langle\theta\rangle \wedge \mathsf{safe}^*(\mathsf{e}) \wedge \neg\langle e\rangle \to \langle\psi\rangle)$ and $A_2 : (\varphi \to \theta \wedge \mathsf{safe}(\mathsf{e})) \wedge (\theta \wedge \mathsf{safe}(\mathsf{e}) \wedge \neg e \to \psi)$

$$(\textsc{seq})\ \dfrac{(\textsc{assert})\ \dfrac{}{\{\langle\varphi\rangle\}\ \textbf{assert}\ \mathsf{safe}^*(\mathsf{e})\ \{\rho\}}\ A_1 \qquad (\textsc{if-else})\ \dfrac{\{\rho \wedge \langle e\rangle\}\ \langle C_1\rangle\ \{\langle\psi\rangle\} \qquad \{\rho \wedge \neg\langle e\rangle\}\ \langle C_2\rangle\ \{\langle\psi\rangle\}}{\{\rho\}\ \textbf{if}\ (\langle e\rangle)\ \{\langle C_1\rangle\}\ \textbf{else}\ \{\langle C_2\rangle\}\ \{\langle\psi\rangle\}}}{\{\langle\varphi\rangle\}\ \textbf{assert}\ \mathsf{safe}^*(\mathsf{e});\ \textbf{if}\ (\langle e\rangle)\ \{\langle C_1\rangle\}\ \textbf{else}\ \{\langle C_2\rangle\}\ \{\langle\psi\rangle\}}$$

$$(\textsc{if-else})\ \dfrac{\{\!|e \wedge \varphi|\!\}\ C_1\ \{\!|\psi|\!\} \qquad \{\!|\neg e \wedge \varphi|\!\}\ C_2\ \{\!|\psi|\!\}}{\{\!|\varphi|\!\}\ \textbf{if}\ (e)\ \{C_1\}\ \textbf{else}\ \{C_2\}\ \{\!|\psi|\!\}}\ A_2$$

where $A_1 : \langle\varphi\rangle \to \mathsf{safe}^*(\mathsf{e}) \wedge \rho$ and $A_2 : \varphi \to \mathsf{safe}(e)$

Figure 7.3: Derivation trees of *While** and *While*^{C} of the proof of Lemma 27

Chapter 8

Conclusions

In this thesis we have studied how language-based techniques, namely the ones based on deductive verification, can be used to provide assurance that the cryptographic software implementations enforce the desired security policies. This chapter highlights the main contribution of this thesis and points out relevant directions for future work.

8.1 Contributions

In the first part of this thesis, we have applied an off-the-shelf verification platform, Frama-c, to verify a specific set of security policies in cryptographic software implementations. Firstly we have shown how safety properties (such as the absence of numeric errors and memory safety), absence of error propagation in stream ciphers and program equivalence, can be verified in an implementation of the OpenSSL library.

The absence of error propagation is formalised as a noninterference property and the *self-composition* technique is applied to prove it. To prove program equivalence we have extended the concept of *self-composition* to a more comprehensive notion, *equivalence by composition*, which allows the formalisation of program equivalence using deductive verification techniques. We also have used the *natural invariants* technique to mechanise these composition-based proofs.

Afterwards, we have addressed a set of high-level non-functional security policies, adopted by the developers of the NaCl library, to enforce software countermeasures against (timing and cache) side-channel attacks. We have formalised resistance to such side-channels attacks as noninterference. The general approach we adopt consists of reifying the target program to make explicit in its output state the execution traces that may potentially leak information. We reduce this explicit information to a minimum, proving that our approach is still sound, and then use noninterference and self-composition to verify security.

In the second part of this thesis we have developed a deductive verification tool for CAO, named CAOVerif. The design of this tool was initiated in collaboration with the ProVal team[1], at INRIA Saclay - Île-de-France research center and continued at Universidade do Minho. The tool relies on the Jessie plug-in of the Frama-c framework as a back-end and essentially translates CAO annotated programs to Jessie. Along with the development of CAOVerif, we have presented a model in first-order logic of certain mathematical objects (present in the CAO language) that have specific interest for cryptography. We believe that the proposed model may be of independent interest and can be of used in other areas, considering that it has been designed to maximise the degree of automation that can be achieved when feeding proof obligations (related to these mathematical abstractions) to general Satisfiability Modulo Theories (SMT) solvers. We have concluded this work by providing a proof of concept of how one can establish the soundness of our deductive verification tool.

8.2 Directions for future work

In addition to showing that deductive verification methods are increasingly more amenable to practical use with reasonable degrees of automation, our work answers some open questions raised by previous work, which seemed to indicate that proofs by (self-)composition were not directly applicable in real-world situations [88]. Our results are promising in that we have been able to achieve our goal using only off-the-shelf verification tools, such as Frama-c, and a technique with a high potential for mechanisation.

However, we should also emphasize that, even though we believe our results show that our approach outperforms previous solutions in the deployment of self-composition proofs, there are still obvious limitations that should be highlighted. The first class of limitations are those inherent to the deductive verification technology itself. For example, for programs displaying high cyclomatic complexity [2], and despite the optimizations

[1]`http://proval.lri.fr/`

[2]Intuitively, programs offering a large number of possible independent execution control-flow paths.

introduced by the existing tools, the number of generated proof obligations tends to increase exponentially. This means that formal verification rapidly becomes impractical. On the other hand, we should also highlight that NaCl code follows strict coding policies that make it *formal verification-friendly*. In particular, it does not use many of the features of the C language that typically complicate matters, including side-effects, pointer casts, or dynamic memory allocation.

In the future we believe that it would be very useful to enhance our tools to allow automating the specification process of the properties like the ones presented in Chapters 4 and 5. We believe that it is possible to automatically apply equivalence by composition and natural invariant techniques, allowing the user to focus only in the proof of a simple lemma. Moreover, it would be interesting to investigate how our methodology can be adapted to prove security policies which are based on relaxed notions of non-interference. These security policies tend to downgrade the security level of a certain piece of information (in the program), and the goal is to prove that the program does not release more information than the one it is allowed.

Another useful improvement would be to adapt the CAOVerif tool to directly generate proof obligations to particular theorem provers, such as *CryptoMiniSat2*[3], in order to automate the proofs that may involve, for example, extended fields or matrix multiplication. Although the extended fields theory proposed in CAOVerif (to model the extension fields type) allow a reasonably automation in the proof of the verification conditions, namely the ones related with safety, it can be much improved. The multi-tiered approach followed in the development of CAOVerif enables the use of proof assistants to do more complicated proofs. However, this may require more expertise and skills than it is desirable. Ideally, we would like to build a tool where the user intervention could be minimal, both at the annotations level, as well as, at the proof level, at least for cryptographic implementations. Investigating how this can be achieved is undoubtedly an interesting research question.

[3]`http://www.msoos.org/cryptominisat2`

Appendix A

Verifying openSSL implementation of RC4

A.1 openSSL implementation of RC4

```
typedef struct rc4_key_st { unsigned char x,y; unsigned char data[256]; } RC4_KEY;

void RC4(RC4_KEY *key,const unsigned long len,  unsigned char *indata,
  unsigned char *outdata) {
  register unsigned char *d; register unsigned char x,y,tx,ty; int i;
  x=key->x; y=key->y;d=key->data;

#define LOOP(in,out) \
  x=((x+1)&0xff); \ tx=d[x]; \ y=((tx+y)&0xff); \ d[x]=ty=d[y]; \ d[y]=tx; \
  (out) = d[((tx+ty)&0xff)]^ (in);

 i=(int)(len>>3L);
 if (i) {
   while(1)  {
     RC4_LOOP(indata,outdata,0); RC4_LOOP(indata,outdata,1);
     RC4_LOOP(indata,outdata,2); RC4_LOOP(indata,outdata,3);
     RC4_LOOP(indata,outdata,4); RC4_LOOP(indata,outdata,5);
     RC4_LOOP(indata,outdata,6); RC4_LOOP(indata,outdata,7);
      indata+=8; outdata+=8; if (--i == 0) break; }
   }

  i=(int)(len&0x07);
  if(i) {
    while(1) {
      RC4_LOOP(indata,outdata,0); if (--i == 0) break;
      RC4_LOOP(indata,outdata,1); if (--i == 0) break;
      RC4_LOOP(indata,outdata,2); if (--i == 0) break;
      RC4_LOOP(indata,outdata,3); if (--i == 0) break;
      RC4_LOOP(indata,outdata,4); if (--i == 0) break;
      RC4_LOOP(indata,outdata,5); if (--i == 0) break;
      RC4_LOOP(indata,outdata,6); if (--i == 0) break;
    }
  }
  key->x=x; key->y=y;
}
```

A.2 ACSL: openSSL RC4 implementation with safety annotations

```
typedef struct rc4_key_st { unsigned char x,y; unsigned char data[256]; } RC4_KEY;

 /*@ lemma valid_range_ax1a: \forall unsigned long k; k>=0 ==> 0<=(k >> 3L)<=k;
   @
   @ lemma valid_range_ax1b:
   @ \forall unsigned long k; k>=0 ==> 0<=((k >> 3L)*8)<=k;
   @
   @ lemma valid_range_ax1c: \forall unsigned long k; k>=0 ==> 0<=(k & 0x07)<=7;
   @
   @ lemma valid_range_ax1d: \forall unsigned long k; k>=0 ==> 0<=(k & 0x07)<= k;
   @
   @ lemma valid_range_ax1e:
   @ \forall unsigned long k; k>=0 ==> ((k >> 3L)*8 + (k & 0x07) == k);
   @
   @ lemma valid_range_ax1f:
   @ \forall unsigned long k; 0<= k < INT_MAX ==> 0<=(k >> 3L)<(INT_MAX / 8);
   @
   @ lemma valid_range_ax1g:
   @ \forall unsigned long k, unsigned long i; 0<= k < INT_MAX &&
   @  0<=i<=(k>>3L) ==> 0<=(((k >> 3L)-i)*8) < INT_MAX;
   @
   @ lemma valid_range_ax2a: \forall unsigned char k; 0 <= (k & 0xff) < 256;
   @
   @ lemma valid_range_ax2b: \forall unsigned char k; 0 <= k < 256;
   @
   @ lemma valid_range_ax2c: \forall unsigned char k; 0 <= ((k + 1) & 0xff) <256;
   @
   @ lemma valid_range_ax2d:
   @ \forall unsigned char k, unsigned char l; 0 <= ((k + l) & 0xff) < 256;
   @
   @ lemma valid_range_ax4: \forall unsigned char inp; \forall unsigned char k;
   @    0 <= (k ^ inp)<256;
   @*/

 /*@ requires len >= 0 && \valid(key) && \valid(key->data + (0..255) ) &&
  @  \valid(indata + (0..(len-1))) && \valid(outdata + (0..(len-1)));
  @*/
void RC4(RC4_KEY *key,const unsigned long len, unsigned char *indata,
    unsigned char *outdata) {
    register unsigned char *d; register unsigned char x,y,tx,ty;
    int i;
    x=key->x; y=key->y; d=key->data;

#define LOOP(in,out) \
   x=((x+1)&0xff); \ tx=d[x]; \ y=((tx+y)&0xff); \ d[x]=ty=d[y]; \ d[y]=tx; \
   (out) = d[((tx+ty)&0xff)]^ (in);
#define RC4_LOOP(a,b,i) LOOP(a[i],b[i])

   i=(int)(len>>3L);
   /*@ ghost int old_i = i;*/
   /*@ ghost unsigned char *old_indata = indata;*/
   /*@ ghost unsigned char *old_outdata = outdata;*/

   //@ ghost goto L;
   //@ ghost L:
  if (i) {
   /*@ loop invariant (0 < i <= (len>>3L)) &&
     @     indata == old_indata + ((old_i - i)*8) &&
     @     outdata == old_outdata + ((old_i - i)*8);
     @ loop variant i;
     @*/
   while(1)  {
    RC4_LOOP(indata,outdata,0); RC4_LOOP(indata,outdata,1);
    RC4_LOOP(indata,outdata,2); RC4_LOOP(indata,outdata,3);
    RC4_LOOP(indata,outdata,4); RC4_LOOP(indata,outdata,5);
    RC4_LOOP(indata,outdata,6); RC4_LOOP(indata,outdata,7);
    indata+=8; outdata+=8; if (--i == 0) break; }
   }
   /*@ assert i==0 && indata == old_indata + ((len >> 3L)*8) &&
     @   outdata == old_outdata +((len >> 3L)*8);
     @*/
  i=(int)(len&0x07);

  if(i) {
   /*@ loop invariant i <=(len&0x07) && i>0;
     @ loop variant i;
     @*/
   while(1) {
    RC4_LOOP(indata,outdata,0); if (--i == 0) break; /*@ assert i>0;*/
    RC4_LOOP(indata,outdata,1); if (--i == 0) break; /*@ assert i>0;*/
    RC4_LOOP(indata,outdata,2); if (--i == 0) break; /*@ assert i>0;*/
    RC4_LOOP(indata,outdata,3); if (--i == 0) break; /*@ assert i>0;*/
    RC4_LOOP(indata,outdata,4); if (--i == 0) break; /*@ assert i>0;*/
    RC4_LOOP(indata,outdata,5); if (--i == 0) break; /*@ assert i>0;*/
    RC4_LOOP(indata,outdata,6); if (--i == 0) break; }
   }
  key->x=x;
  key->y=y;
}
```

Appendix B

NaCl implementation of crypto_verify

B.1 Equivalence by composition

Refactoring step 1

```
/*@ predicate body_loop1{L1,L2}( unsigned char *x,
 @ unsigned char *y, integer i1, integer i2,integer res1, integer res2) =
 @ i2 == i1 + 1 && ( (\at(x[i1],L1) != \at(y[i1],L1) && res2 == -1) ||
 @ (\at(x[i1],L1) == \at(y[i1],L1) && res2 == res1));
 @*/

/*@ inductive loop_pred{L1,L2}(integer i1,integer i2, unsigned char *x,
 @ unsigned char *y,integer res1, integer res2){
 @ case base_case{L}: \forall unsigned char *x,*y; \forall integer i,res;
 @ loop_pred{L,L}(i,i,x,y,res,res);
 @ case ind_case_if{L1,L2,L3}:
 @ \forall unsigned char *x,*y; \forall integer i1,i2,i3,res1,res2,res3;
 @ loop_pred{L1,L2}(i1,i2,x,y,res1,res2) ==>
 @ body_loop1{L2,L3}(x,y,i2,i3,res2, res3) ==>
 @ loop_pred{L1,L3}(i1,i3,x,y,res1,res3);
 @ }
 @*/

/*@ predicate body_loop3{L1,L2}( unsigned char *x, unsigned char *y,
 @  integer i1, integer i2,integer diffbits1 , integer diffbits2) =
 @  i2 == i1+1 && diffbits2 == (diffbits1 | (\at(x[i1],L1) ^ \at(y[i1],L1)));
 @*/

/*@ inductive loop_pred3{L1,L2}(integer i1,integer i2, unsigned char *x,
 @ unsigned char *y, integer diffbits1, integer diffbits2){
 @ case base_case{L}: \forall unsigned char *x,*y; \forall integer i,diffbits;
 @  loop_pred3{L,L}(i,i,x,y,diffbits,diffbits);
 @ case ind_case{L1,L2,L3}: \forall unsigned char *x,*y;
 @  \forall integer i1,i2,i3,diffbits1,diffbits2,diffbits3;
 @  loop_pred3{L1,L2}(i1,i2,x,y,diffbits1 ,diffbits2) ==>
 @       body_loop3{L1,L2}(x,y,i2,i3,diffbits2 ,  diffbits3) ==>
 @       loop_pred3{L1,L3}(i1,i3,x,y,diffbits1 ,diffbits3);
 @ }
 @*/

/*@ lemma eq_loops{L1,L2,L3,L4}: \forall integer i; \forall int bits1, res;
 @ \forall unsigned char *x,*y, *x1, *y1;
 @ \forall integer j; \at(x[j],L1)==\at(x1[j],L3) ==>
 @ \forall integer j; \at(y[j],L1)==\at(y1[j],L3) ==>
 @ loop_pred{L1,L2}(0,i,x,y,0,res) ==> loop_pred3{L3,L4}(0,i,x1,y1,0,bits1) ==>
 @        ((res ==0 && bits1==0) || (res ==-1 && bits1!=0));
 @*/

/*@ requires (\forall integer j; 0<=j<16 ==> x[j]==x1[j]) &&
 @       (\forall integer j; 0<=j<16 ==> y[j]==y1[j]);
 @*/
void crypto_verify(const unsigned char *x, const unsigned char *y,
 const unsigned char *x1, const unsigned char *y1, int res, int res1) {
 int i, i1; int differentbits1 = 0; i = 0; i1 = 0; res = 0; res1 = 0;

 //@ ghost goto L1;
 //@ ghost L1:

 /*@ loop invariant 0<=i<=16 && loop_pred{L1,Here}(0,i,x,y,0,res);
   @ loop variant 16 i;
   @*/
 while (i < 16) { if (x[i] != y[i]) { res = -1; } i++;}

 //@ ghost goto L2;
 //@ ghost L2:

 /*@ loop invariant 0<=i1<=16 &&
   @  loop_pred3{L2,Here}(0,i1,x1,y1,0,differentbits1);
   @ loop variant 16-i1;
   @*/
 while (i1 < 16) { differentbits1 |= x1[i1] ^ y1[i1]; i1++;}
 if (differentbits1 != 0) res1 = -1; //@ assert res==res1;
}
%\end{ccode}
```

Refactoring step 2

```
/*@ predicate body_loop3{L1,L2}( unsigned char *x, unsigned char *y,
 @ integer i1, integer i2,integer diffbits1 , integer diffbits2) = i2==i1 +1 &&
 @ diffbits2 == (diffbits1 | (\at(x[i1],L1) ^ \at(y[i1],L1)));
 @*/

/*@ inductive loop_pred3{L1,L2}(integer i1,integer i2, unsigned char *x,
 @   unsigned char *y,integer diffbits1, integer diffbits2){
```

```
 @ case base_case{L}: \forall unsigned char *x,*y;
 @ \forall integer i,diffbits;loop_pred3{L,L}(i,i,x,y,diffbits,diffbits);
 @ case ind_case{L1,L2,L3}: \forall unsigned char *x,*y;
 @ \forall integer i1,i2,i3,diffbits1,diffbits2,diffbits3;
 @ loop_pred3{L1,L2}(i1,i2,x,y,diffbits1,diffbits2) ==>
 @ body_loop3{L1,L2}(x,y,i2,i3,diffbits2, diffbits3) ==>
 @ loop_pred3{L1,L3}(i1,i3,x,y,diffbits1,diffbits3);
 @ }
 @*/

/*@ lemma eq_loops{L1,L2,L3,L4}:
 @ \forall integer i; \forall int bits1, bits2;
 @ \forall unsigned char *x,*y, *x1, *y1;
 @ \forall integer j; \at(x[j],L1)==\at(x1[j],L3) ==>
 @ \forall integer j; \at(y[j],L1)==\at(y1[j],L3) ==>
 @ loop_pred3{L1,L2}(0,i,x,y,0,bits1) ==>loop_pred3{L3,L4}(0,i,x1,y1,0,bits2)==>
 @ \at(bits1,L2) == \at(bits2,L4);
 @*/

/*@ lemma diffbits_eq_zero:
  @ \forall int bits; bits ==0 ==> (1&((bits-1)>>8))-1==0;
  @*/

/*@ lemma diffbits_neq_zero:
  @ \forall int bits; bits !=0 ==> (1&((bits-1)>>8))-1==-1;
  @*/

/*@ requires (\forall integer j; 0<=j<16 ==> x[j]==x1[j]) &&
 @       (\forall integer j; 0<=j<16 ==> y[j]==y1[j]);
 @*/
void crypto_verify(const unsigned char *x, const unsigned char *y,
 const unsigned char *x1, const unsigned char *y1, int res, int res1) {

 int i, i1; int differentbits = 0; int differentbits1 = 0;
 i = 0; i1 = 0;res = 0; res1 = 0;

        //@ ghost goto L1;
        //@ ghost L1:

   /*@ loop invariant 0<=i<=16 &&
     @ loop_pred3{L1,Here}(0,i,x,y,0,differentbits);
        @ loop variant 16-i;
        @*/
        while (i < 16) { differentbits |= x[i] ^ y[i]; i++; }
        if (differentbits != 0) res = -1;
        //@ ghost goto L2;
        //@ ghost L2:

   /*@ loop invariant 0<=i1<=16 &&
     @ loop_pred3{L2,Here}(0,i1,x1,y1,0,differentbits1);
        @ loop variant 16-i1;
        @*/
        while (i1 < 16) { differentbits1 |= x1[i1] ^ y1[i1]; i1++; }
        res1 = (1& ((differentbits1-1)>>8))-1;
        //@ assert res==res1;
}
```

Specification of the refactoring step 3

```
/*@ predicate body_loop3{L1,L2}( unsigned char *x, unsigned char *y,
 @ integer i1, integer i2,integer diffbits1 , integer diffbits2)=i2 == i1 +1 &&
 @ diffbits2 == (diffbits1 | (\at(x[i1],L1) ^ \at(y[i1],L1)));
 @*/

/*@ inductive loop_pred3{L1,L2}(integer i1,integer i2, unsigned char *x,
 @ unsigned char *y, integer diffbits1 , integer diffbits2){
 @ case base_case{L}: \forall unsigned char *x,*y;
 @ \forall integer i,diffbits;loop_pred3{L,L}(i,i,x,y,diffbits ,diffbits);
 @ case ind_case{L1,L2,L3}:
 @ \forall unsigned char *x,*y;
 @ \forall integer i1,i2,i3,diffbits1 ,diffbits2 ,diffbits3;
 @ loop_pred3{L1,L2}(i1,i2,x,y,diffbits1 ,diffbits2) ==>
 @ body_loop3{L1,L2}(x,y,i2,i3,diffbits2 , diffbits3) ==>
 @ loop_pred3{L1,L3}(i1,i3,x,y,diffbits1 ,diffbits3);
 @ }
 @*/

/*@ lemma eq_loop_pred_false{L1,L2,L3,L4}:
 @ \forall integer i; \forall int bits, bits1; \forall unsigned char *x,*y;
 @ \forall unsigned char *x1,*y1;
 @ \forall integer j; \at(x[j],L1)==\at(x1[j],L3) ==>
 @ \forall integer j; \at(y[j],L1)==\at(y1[j],L3) ==>
 @ loop_pred3{L1,L2}(0,i,x,y,0,bits) ==>loop_pred3{L3,L4}(0,i,x1,y1,0,bits1)==>
 @ \at(bits,L2) == \at(bits1,L4);
 @*/
%\end{ccode}
```

B.2 Annotated self-composed crypto_verify function

```
/*@ predicate body{L1,L2}(unsigned char *x,unsigned char *y,
  @ integer diffbits1, integer diffbits2, list l1x, list l2x, list l1y, list l2y,
  @ list l1ctrl, list l2ctrl, integer i1, integer i2) =
  @ i2==i1+1 && (diffbits2==(diffbits1|(\at(x[i1],L1)^\at(y[i1],L1)))) &&
  @ l2ctrl==cons(i2<16?1:0,l1ctrl) && l2x==cons(i1,l1x) && l2y==cons(i1,l1y);@*/

/*@ inductive loop_pred{L1,L2}(integer i1, integer i2, unsigned char *x,
  @ unsigned char *y, integer diffbits1, integer diffbits2, list l1_x, list l2_x,
  @ list l1_y, list l2_y,list l1_control, list l2_control){
  @  case base_case{L}:
```

```
  @   \forall list lx, ly, lcontrol, integer i, diffbits, unsigned char *x,*y;
  @    loop_pred{L,L}(i,i,x,y,diffbits,diffbits, lx,lx,ly,ly,lcontrol,lcontrol);
  @  case ind_case{L1,L2,L3}:
  @   \forall unsigned char *x,*y, list l1_x, l2_x, l3_x, l1_y, l2_y, l3_y;
  @   list l1_control, l2_control, l3_control, integer i1,i2,i3;
  @   integer diffbits1, diffbits2, diffbits3;
  @   loop_pred{L1,L2}(i1, i2, x, y, diffbits1, diffbits2, l1_x, l2_x,
  @                    l1_y, l2_y, l1_control, l2_control) ==>
  @   body{L2,L3}(x, y, diffbits2, diffbits3, l2_x, l3_x,
  @                l2_y, l3_y, l2_control, l3_control,i2,i3) ==>
  @   loop_pred{L1,L3}(i1, i3, x, y, diffbits1, diffbits3, l1_x, l3_x,
  @                    l1_y, l3_y, l1_control, l3_control); } @*/

/*@ requires lmem_control == lmem_control1
  @         && lmem_x == lmem_x1 && lmem_y == lmem_y1;
  @ ensures lmem_control == lmem_control1
  @         && lmem_x == lmem_x1 && lmem_y == lmem_y1;  @*/
void crypto_verify(const unsigned char *x, const unsigned char *y,
 const unsigned char *x1, const unsigned char *y1, int result, int result1) {
   int differentbits = 0, differentbits1 = 0, i = 0, i1 = 0;

   /*@ ghost append_control(i<16);
     @ ghost L1:
     @ loop invariant 0<=i<=16 &&
     @ loop_pred{L1,Here}(0,i,x,y,0,differentbits,lmem_x{L1},lmem_x,
     @                    lmem_y{L1},lmem_y,lmem_control{L1},lmem_control); @*/
   while (i < 16) { F(i) //@ ghost append_x(i); ghost append_y(i);
      i++; //@ ghost append_control(i<16); }
   result = (1 & ((differentbits - 1) >> 8)) - 1;
   /*@ ghost append_control1(i1<16);
     @ ghost L2:
     @ loop invariant 0<=i1<=16 &&
     @ loop_pred{L2,Here}(0,i1,x1,y1,0,differentbits1,lmem_x1{L2},lmem_x1,
     @               lmem_y1{L2},lmem_y1,lmem_control1{L2},lmem_control1);@*/
   while (i1 < 16) { F1(i1) //@ ghost append_x1(i1); ghost append_y1(i1);
      i1++; //@ ghost append_control1(i1<16); }
   result1 = (1 & ((differentbits1 - 1) >> 8)) - 1;
}
```

Appendix C

CAO implementation of crypto_scalar_mult

```
typedef byte      := unsigned bits[8];
typedef unpacked  := unsigned bits[256];
typedef packed    := vector[32] of unsigned bits[8];
typedef skey      := unsigned bits[255];
typedef Fp        := mod[2**255-19];

/* Curve points in Montgomery representation */
typedef MontRep   := struct [ def x : Fp; def z : Fp; ];

/*@ lemma is_prime_Fp: is_prime(2**255-19); */

/* Constant global curve parameter */
def a2 : Fp := [486662];
/*@ global invariant constant_a2: a2==[486662] */

/* Curve point addition */
def addMont(Q,Qpr,QmQpr : MontRep) : MontRep {
    def Q3 : MontRep;

    Q3.x := [4] * (Q.x * Qpr.x - Q.z*Qpr.z)**2 * QmQpr.z;
    Q3.z := [4] * (Q.x * Qpr.z - Q.z*Qpr.x)**2 * QmQpr.x;

    return Q3;
}

/* Curve point addition */
def doubleMont(Q : MontRep) : MontRep {
    def Q2 : MontRep;

    Q2.x := (Q.x**2 - Q.z**2)**2;
    Q2.z := [4]*Q.x*Q.z*(Q.x**2+a2*Q.x*Q.z+Q.z**2);

    return Q2;
}

/* Curve point scalar multiplication */
def curve25519(n : skey, base : Fp) : Fp {
    def i : int := 253;
    def mth, mp1th, one : MontRep;

    one.x := base;
    one.z := [1];
    mth := one;
    mp1th := doubleMont(one);

    /*@ invariant -1<=i<=253
        variant i */
    while(i>=0) {
        if (n[i] == 1) {
           mth := addMont(mth,mp1th,one);
           mp1th := doubleMont(mp1th);
        }
        else {
           mp1th := addMont(mth,mp1th,one);
           mth := doubleMont(mth);
        }
        i := i-1;
    }
    if (mth.z == [0]) {
        return [0];
    }
    else {
        return (mth.x/mth.z);
    }
}
/* Unpacking a byte array */
/*@ ensures (forall i,j,k:int;
        (0<=i<32 && 0<=j<8 && k==i*8+j) ==> in[i][j]==result[k]) */
def unpack(in : packed) : unpacked {
    return (in[ 0] @ in[ 1] @ in[ 2] @ in[ 3] @ in[ 4] @ in[ 5] @
            in[ 6] @ in[ 7] @ in[ 8] @ in[ 9] @ in[10] @ in[11] @
            in[12] @ in[13] @ in[14] @ in[15] @ in[16] @ in[17] @
            in[18] @ in[19] @ in[20] @ in[21] @ in[22] @ in[23] @
            in[24] @ in[25] @ in[26] @ in[27] @ in[28] @ in[29] @
            in[30] @ in[31]);
}

/* Reconstructing the secret key */
/*@ ensures result[0]== 0b0 && result[1  ]==0b0 &&
            result[2]== 0b0 && result[254] == 0b1
    ensures (forall i,j:int;
            0<=i<32 && 0<=j<8 && 2<i*8+j<254 ==>
            n[i][j]==result[i*8+j])
 */
```

```
def clampC(n : packed) : skey {
   def key: skey;
   def pack : unpacked;

   pack := unpack(n);

   pack[0..2] := 0b000;
   /*@ assert pack[0]==0b0 */
   /*@ assert pack[1]==0b0 */
   /*@ assert pack[2]==0b0 */

   pack[254..255]  := 0b01;
   /*@ assert pack[254]==0b1 */
   /*@ assert pack[255]==0b0 */

   /*@ assert (forall i,j:int;
         0<=i<32 && 0<=j<8 && 2<i*8+j<254 ==> n[i][j] == pack[i*8+j])*/

    key := pack[0..254];
    /*@ assert (forall k:int; 0<=k<=254 ==> key[k] == pack[k])*/
    /*@ assert (forall i,j:int;
         0<=i<32 && 0<=j<8 && 2<i*8+j<254 ==> n[i][j] == key[i*8+j])*/
    return key;
}

/* Packing a byte array */
def pack(in : unpacked) : packed {
    def out : packed;

    seq i := 0 to 31 {
        out[i]:=in[8*i..8*i+7];
    }

    return out;
}

/* Entry point */
def crypto_scalarmult(n,p : packed) : packed {
    def pm : Fp := (Fp)unpack(p);
    def nc : skey := clampC(n);

    return(pack((unpacked)(int)curve25519(nc,pm)));
}
```

Appendix D

Cao to Jessie translation

The translation of CAO to Jessie is denoted by $\langle x \rangle$, where x can be any part of the input program, e.g. a full CAO program, an expression, an annotation, etc.

D.1 Expressions

Literals

$$\langle \mathbf{true} \rangle = \mathbf{true} \quad \langle 0b1 \rangle = bits\_set(bits\_null\_vector, 0, true) \quad \langle [i] \rangle = mod\_\langle \tau \rangle\_of\_int(i)$$
$$\langle \mathbf{false} \rangle = \mathbf{false} \quad \langle 0b0 \rangle = bits\_set(bits\_null\_vector, 0, false) \quad \langle i \rangle = i$$

$$\begin{aligned} \langle \{e_1, ..., e_n\} \rangle = {} & \mathbf{let}\ v_1 = vector\_\langle \tau \rangle\_set(vector\_\langle \tau \rangle\_any, 0, \langle e_1 \rangle) \\ & \mathbf{in\ let}\ v_2 = vector\_\langle \tau \rangle\_set(v_1, 0, \langle e_2 \rangle)\ \mathbf{in}\ ....\ vector\_\langle \tau \rangle\_set(v_{n-1}, n, \langle e_n \rangle) \end{aligned}$$

Integer and boolean expressions

$$\langle x \rangle = x \qquad \langle -e \rangle = -\langle e \rangle \qquad \langle i_1\ op\ i_2 \rangle = \langle i_1 \rangle\ op\ \langle i_2 \rangle$$
$$\langle !b \rangle = !\langle b \rangle \qquad \langle f_1\ op\ f_2 \rangle = int\_of\_mod\_\langle \tau \rangle(\langle f_1 \rangle\ op\ \langle f_2 \rangle)$$

Vectors and matrices expressions

$$\langle v[i] \rangle = \mathbf{let}\ x = \langle i \rangle\ \mathbf{in\ assert}\ 0 \leq x < n\ vector\_\langle \tau \rangle\_get(\langle v \rangle, x)$$

$$\begin{aligned}
\langle v[i_1..i_2]\rangle &= \textbf{let } x_1 = \langle i_1\rangle \textbf{ in let } x_2 = \langle i_2\rangle \textbf{ in assert } 0 \le x_1 < n \wedge 0 \le x_2 < n\\
&\quad vector\_\langle\tau\rangle\_shift(\langle v\rangle, x_)\\
\langle m[i_1, i_2]\rangle &= \textbf{let } x_1 = \langle i_1\rangle \textbf{ in let } x_2 = \langle i_2\rangle \textbf{ in assert } 0 \le x_1 < n_1 \wedge 0 \le x_2 < n_2\\
&\quad matrix\_\langle\tau\rangle\_get(\langle m\rangle, x_1, x_2)\\
\langle m[i_1..i_2, i_3..i_4]\rangle &= \textbf{let } x_1 = \langle i_1\rangle \textbf{ in let } x_2 = \langle i_2\rangle \textbf{ in let } x_3 = \langle i_3\rangle \textbf{ in let } x_4 = \langle i_4\rangle \textbf{ in}\\
&\quad \textbf{assert } 0 \le x_1 < n_1 \wedge 0 \le x_2 < n_1 \wedge 0 \le x_3 < n_2 \wedge 0 \le x_4 < n_2\\
&\quad matrix\_\langle\tau\rangle\_shift(\langle m\rangle, x_1, x_3)
\end{aligned}$$

Unsigned bits expressions

$$\begin{aligned}
\langle u[i]\rangle &= \textbf{let } x = \langle i\rangle \textbf{ in assert } 0 \le x < n;\ bits\_get(\langle u\rangle, x)\\
\langle u[i_1..i_2]\rangle &= \textbf{let } x_1 = \langle i_1\rangle \textbf{ in let } x_2 = \langle i_2\rangle \textbf{ in assert } 0 \le x_1 < n \wedge 0 \le x_2 < n\\
&\quad bits\_shift(\langle u\rangle, x_1)
\end{aligned}$$

$$\begin{aligned}
\langle \sim u\rangle &= bits\_bitwise\_neg(\langle u\rangle) & \langle u_1 \mathbin{\hat{}} u_2\rangle &= bits\_bitwise\_xor(\langle u_1\rangle, \langle u_2\rangle)\\
\langle u_1 \,\&\, u_2\rangle &= bits\_bitwise\_and(\langle u_1\rangle, \langle u_2\rangle) & \langle u_1 \mid u_2\rangle &= bits\_bitwise\_or(\langle u_1\rangle, \langle u_2\rangle)
\end{aligned}$$

$$\begin{aligned}
\langle u \;>>\; i\rangle &= bits\_blit(bits\_null\_vector, bits\_shift(\langle u\rangle, \langle i\rangle), len - \langle i\rangle, \langle i\rangle)\\
\langle u \;<<\; i\rangle &= bits\_blit(bits\_shift(\langle u\rangle, 0), bits\_null\_vector, \langle i\rangle, len - \langle i\rangle)\\
\langle u \mid> i\rangle &= bits\_blit(bits\_shift(\langle u\rangle, 0), bits\_shift(\langle u\rangle, \langle i\rangle), (len - \langle i\rangle), \langle i\rangle)\\
\langle u \;<\mid i\rangle &= bits\_blit(bits\_shift(\langle u\rangle, 0), bits\_shift(\langle u\rangle, len - \langle i\rangle), \langle i\rangle, len - \langle i\rangle)
\end{aligned}$$

General expressions

$$\langle x\rangle = x \quad , x \in \mathsf{Var}_\tau \qquad\qquad \langle f(e_1, \ldots, e_n)\rangle = \langle f\rangle(\langle e_1\rangle, \ldots, \langle e_n\rangle)$$

D.2 Statements

Variable assignment

$$\begin{aligned}
\langle x := e\rangle &= x = \langle e\rangle \\
\langle l[i] := e\rangle &= l = \textbf{let } x = \langle i\rangle \textbf{ in assert } 0 \le x < n\ vector\_\langle\tau\rangle\_set(l, x, \langle e\rangle) \\
\langle l[i_1..i_2] := e\rangle &= \textbf{let } x_1 = \langle i_1\rangle \textbf{ in let } x_2 = \langle i_2\rangle \textbf{ in assert } 0 \le x_1 < n \wedge 0 \le x_2 < n \\
& \quad l = vector\_\langle\tau\rangle\_blit(\langle e\rangle, \langle l[i_1..i_2]\rangle, x_1, succ(x_2 - x_1)) \\
\langle l[i_1, i_2] := e\rangle &= \textbf{let } x_1 = \langle i_1\rangle \textbf{ in let } x_2 = \langle i_2\rangle \textbf{ in assert } 0 \le x_1 < n_1 \wedge 0 \le x_2 < n_2 \\
& \quad l = matrix\_\langle\tau\rangle\_set(l, x_1, x_2, \langle e\rangle)
\end{aligned}$$

Multiple assignment

$$\begin{aligned}
\langle e_1, e_2, ..., e_n := e'_1, e'_2, ..., e'_n\rangle = {} & \textbf{var } \tau_1 \, \mathsf{y}_1 = \langle e'_1\rangle;\ \textbf{var } \tau_2 \, \mathsf{y}_2 = \langle e'_2\rangle;\ ...\ \textbf{var } \tau_n \, \mathsf{y}_n = \langle e'_n\rangle; \\
& \langle e_1 := y_1;\ e_2 := y_2;\ ...\ e_n := y_n; \rangle
\end{aligned}$$

for $e_1 : \tau_1, \ldots,\ e_n : \tau_n$ and $y_1, \ldots, y_n$ (new) fresh variables.

Assigning to a function that returns more than one value

$$\begin{aligned}
& \langle e_1, e_2, ..., e_n := f(e_1, ..., e_n)\rangle = \\
& \quad \textbf{let } \mathsf{struct\_}\tau\ y = \langle f(x_1, ..., x_n)\rangle \textbf{ in } \{\langle e_1\rangle = struct\_\tau\_get\_field_1(y); \\
& \quad \langle e_2\rangle = struct\_\tau\_get\_field_2(y);\ ...\ \langle e_n\rangle = struct\_\tau\_get\_field_n(y); \}
\end{aligned}$$

For $f(e_1, .., e_n) : \mathsf{struct\_}\tau$.

Definitions and initialisations

$$\begin{aligned}
\langle \textbf{def } a : vector[n]\ of\ \tau\rangle &= \textbf{var } \langle vector[n]\ of\ \tau\rangle\ a = \langle vector[n]\ of\ \tau\rangle\_any() \\
\langle \textbf{def } a : bits[n]\rangle &= \textbf{var } bits\ a = bits\_any() \\
\langle \textbf{def } a : matrix[n_1, n_2]\ of\ \tau\rangle &= \textbf{var } \langle matrix[n_1, n_2]\ of\ \tau\rangle\ a = \langle matrix[n_1, n_2]\ of\ \tau\rangle\_any() \\
\langle \textbf{def } x : \tau\rangle &= \textbf{var } \langle\tau\rangle\ x \\
& \quad \text{for } \tau \notin \{\text{vector[n] of } \tau, \text{bits[n]}, \text{matrix[n}_1, \text{n}_2\text{] of } \tau\} \\
\langle \textbf{def } x : \tau := e\rangle &= \textbf{var } \langle\tau\rangle\ x = \langle e\rangle \\
\langle \textbf{def } x : \tau := \{e_1, ..., e_n\}\rangle &= \textbf{var } \langle\tau\rangle\ x = \langle\tau\rangle\_any();\ x = \langle\tau\rangle\_set(x, 0, \langle e_1\rangle);\ ... \\
& \quad x = \langle\tau\rangle\_set(x, n, \langle e_n\rangle);
\end{aligned}$$

Function and procedure arguments

$\langle x \rangle = x\_input$ for $type(x) \in \{vector[n] \cup bits[n] \cup matrix[n_1, n_2]\}$ and $x \in Var_\tau$

$\langle x \rangle = x$ for $type(x) \notin \{vector[n] \cup bits[n] \cup matrix[n_1, n_2]\}$ and $x \in Var_\tau$

Commands

$\langle \textbf{if}\ (b)\ \{c\} \rangle = \textbf{if}\ \langle b \rangle\ \textbf{then}\ \langle c \rangle$ $\quad$ $\langle \textbf{if}\ (b)\ \{c_1\}\ \textbf{else}\ \{c_2\} \rangle = \textbf{if}\ \langle b \rangle\ \textbf{then}\ \langle c_1 \rangle\ \textbf{else}\ \langle c_2 \rangle$

$\langle \textbf{while}\ (b)\ \{c\} \rangle = \textbf{loop while}\ \langle b \rangle\ \langle c \rangle$ $\quad$ $\langle f(e_1, \ldots, e_n) \rangle = \langle f \rangle(\langle e_1 \rangle, \ldots, \langle e_n \rangle)$

$\langle \{C_1; \ldots; C_n;\} \rangle = \{\langle C_1 \rangle; \ldots; \langle C_n \rangle;\}$ $\quad$ $\langle \textbf{return}\ e \rangle = \textbf{return}\ \langle e \rangle$

$$
\begin{aligned}
\langle \textbf{seq}\ x := e_1\ \mathit{to}\ e_2\ \{C_1; ...C_n;\} \rangle &= \textbf{let}\ y_1 = \langle e_1 \rangle\ \textbf{in}\ \{\langle C_1 \rangle[y_1/x]; \ldots \langle C_n \rangle[y_1/x];\} \\
&\quad \textbf{let}\ y_2 = \langle e_1 \rangle + 1\ \textbf{in}\ \{\langle C_1 \rangle[y_2/x]; \ldots \langle C_n \rangle[y_2/x];\} \\
&\quad ... \\
&\quad \textbf{let}\ y_n = \langle e_2 \rangle\ \textbf{in}\ \{\langle C_1 \rangle[y_n/x]; ...\langle C_n \rangle[y_n/x];\} \\
\langle \textbf{return}\ e_1, ..., e_n \rangle &= \textbf{var}\ \mathsf{struct\_}\tau\ x;\quad x = \mathsf{struct\_}\tau\mathsf{\_set\_field}_1(x, \langle e_1 \rangle); \\
&\quad x = \mathsf{struct\_}\tau\mathsf{\_set\_field}_2(x, \langle e_2 \rangle);\ \ldots \\
&\quad x = \mathsf{struct\_}\tau\mathsf{\_set\_field}_n(x, \langle e_n \rangle); \\
&\quad \textbf{return}\ \ x;
\end{aligned}
$$

And $\mathsf{struct\_}\tau$ is struct that represents the return type of the function.

D.3 Annotations

Annotations

$\langle \textbf{invariant}\ \phi \rangle = \textbf{loop invariant}\ \langle \phi \rangle$ $\quad$ $\langle \textbf{variant}\ \phi \rangle = \textbf{loop variant}\ \langle \phi \rangle$

$\langle \textbf{assert}\ \phi \rangle = \textbf{assert}\ \langle \phi \rangle$ $\quad$ $\langle \textbf{label}\ L \rangle = \textbf{label}\ \langle L \rangle$

Assertions

$$\langle \textbf{true} \rangle = \textbf{true} \qquad \langle \textbf{false} \rangle = \textbf{false}$$

$$\langle \phi \wedge \psi \rangle = \langle \phi \rangle \wedge \langle \psi \rangle \qquad \langle e_1 == e_2 \rangle = \langle e_1 \rangle ==_{\langle \tau \rangle} \langle e_2 \rangle \text{ for } e_1, e_2 : \tau$$

$$\langle \phi \vee \psi \rangle = \langle \phi \rangle \vee \langle \psi \rangle \qquad \langle \phi \rightarrow \psi \rangle = \langle \phi \rangle \rightarrow \langle \psi \rangle$$

$$\langle \phi \leftrightarrow \psi \rangle = \langle \phi \rangle \leftrightarrow \langle \psi \rangle \qquad \langle p(e_1, \ldots, e_n) \rangle = \langle p \rangle(\langle e_1 \rangle, \ldots, \langle e_n \rangle)$$

$$\langle \exists\, x : \tau.\ \phi \rangle = \exists\, x : \langle \tau \rangle.\ \langle \phi \rangle \qquad \langle \forall\, x : \tau.\ \phi \rangle = \forall\, x : \langle \tau \rangle.\ \langle \phi \rangle$$

$$\langle \textbf{result} \rangle = \backslash result \qquad \langle \textbf{at}(e, L) \rangle = \backslash at(\langle e \rangle, L)$$

$$\langle \textbf{old}(e) \rangle = \backslash old(\langle e \rangle)$$

$$\langle p\{L_1, ..., L_n\}(x_1, ..., x_n) \rangle = \langle p \rangle\{\langle L_1 \rangle, \ldots, \langle L_n \rangle\}(\langle x_1 \rangle, ..., \langle x_n \rangle)$$

where p is a predicate identifier.

Inductive predicates

$$\langle \textbf{inductive}\ p\{L_1, ..., L_n\}(x_1, ..., x_n)\ \{\textbf{case}\ c_1\{L_1, ..., L_n\} :\ \phi_1\ ...\ \textbf{case}\ c_n\{L_1, ..., L_n\} :\ \phi_n\} \rangle =$$
$$\textbf{logic}\ p\{L_1, ..., L_n\}(\langle x_1 \rangle, ..., \langle x_n \rangle)\{\textbf{case}\ c_1\{L_1, ..., L_n\} :\ \langle \phi_1 \rangle\ ...\ \textbf{case}\ c_n\{L_1, ..., L_n\} : \langle \phi_n \rangle\}$$
$$\langle \textbf{lemma}\ f\{L_1, ..., L_n\} :\ \phi \rangle\ =\ \textbf{lemma}\ f\{L_1, ..., L_n\} : \langle \phi \rangle$$

Logic predicates and logic functions

$$\langle \textbf{predicate}\ p\{L_1, ..., L_n\}(x_1, ..., x_n)\ =\ \phi \rangle\ =\ \textbf{logic}\ p\{L_1, ..., L_n\}(\langle x_1 \rangle, ..., \langle x_n \rangle)\ =\ \langle \phi \rangle$$
$$\langle \textbf{logic}\ \tau\ p\{L_1, ..., L_n\}(x_1, ..., x_n)\ =\ e \rangle\ =\ \textbf{logic}\ \langle \tau \rangle\ p\{L_1, ..., L_n\}(\langle x_1 \rangle, ..., \langle x_n \rangle)\ =\ \langle e \rangle$$

Logic variables and types

$$\langle \textbf{logic}\ \tau\ t \rangle\ = \textbf{logic type}\ \langle \tau \rangle\ t \qquad \langle \textbf{logic def}\ v :\ \tau \rangle\ = \textbf{logic}\ \langle \tau \rangle\ v$$

Axiomatic definition

$$\langle \textbf{axiomatic}\ a\ \{A_1; ...A_n;\} \rangle\ =\ \textbf{axiomatic}\ a\{\langle A_1 \rangle ... \langle A_n \rangle\}$$

D.4 Global declarations

Function/Procedure declaration When the function only returns one type, it is translated as follows:

$$\langle \textbf{def}\, f(x_1 : \tau_1, \ldots, x_n : \tau_n) : \tau\{C\}\rangle \;=\; \langle\tau\rangle\, f(\langle x_1\rangle : \langle\tau_1\rangle, \ldots, \langle x_n\rangle : \langle\tau\rangle_n)\;\{\langle C\rangle;\}$$

and the functions/procedure inputs are translated as:

$$\begin{aligned}
\langle x :\; \tau\rangle \;&\equiv\; input\_x : \langle\tau\rangle \\
&\qquad \tau \in \{bits[n], vector[n], matrix[n_1, n_2], mod[n][X]/f(X), \textbf{struct}\,\rho\} \\
\langle x :\; \tau\rangle \;&\equiv\; x : \langle\tau\rangle \\
&\qquad \tau \notin \{bits[n], vector[n], matrix[n_1, n_2], mod[n][X]/f(X), \textbf{struct}\,\rho\}
\end{aligned}$$

When it is the case that the input variables are translated as *input_x*, then are added to the function's body initializations of the form **var** $\langle\tau\rangle\; x = input\_x$; for $\tau \in \{bits[n], vector[n], matrix[n_1, n_2], mod[n][X]/f(X)\}$, and of the form

$$\textbf{var}\;\langle\tau\rangle[0]\; x = \textbf{new}\;\langle\tau\rangle[1];\;\; \langle\tau\rangle\_copy(input\_x, x);$$

for $\tau \in \{\textbf{struct}\;\rho\}$. For functions declaration whose return type is a tuple of elements we have:

$$\langle \textbf{def}\, f(x_1 : \tau_1, ..., x_n : \tau_n) : \sigma_1, ..., \sigma_n\;\{C\}\rangle = \;f(x_1 : \tau_1, ..., x_n : \tau_n)\;\{\langle C\rangle;\}$$

where $\gamma = declare\_struct(\sigma_1, ..., \sigma_n)$. This means that for each function whose the return type is a tuple of elements is created a struct where each field has the same type of each element present in the tuple and with the same order.

Global variable declaration

$$\langle \textbf{def}\, x : \tau\rangle \;\equiv\; \langle\tau\rangle\; x \qquad \langle \textbf{def}\, x : \tau := e\rangle \;\equiv\; \langle\tau\rangle\; x;\; \textbf{unit}\; global\_var\_init\_x()\{\langle x := e\rangle\}$$

Function contracts and Ghost code

$$\langle \textbf{requires}\;\phi\rangle \;= \textbf{requires}\;\langle\phi\rangle \quad \langle \textbf{ensures}\;\phi\rangle \;= \textbf{ensures}\;\langle\phi\rangle \quad \langle \textbf{ghost}\;\; p\rangle \;=\!\textbf{ghost}\;\langle p\rangle$$

Appendix E

CAO implementation of AES

```
typedef GF2  := mod[ 2 ];
typedef GF2N := mod[ GF2<X> / X**8 + X**4 + X**3 + X + 1 ];
typedef GF2V := vector[8] of GF2;

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

typedef Row  := matrix[1,4] of GF2N;
typedef RowV := vector[4] of GF2N;
typedef Col  := matrix[4,1] of GF2N;
typedef ColV := vector[4] of GF2N;

typedef Byte := unsigned bits[8];
typedef Bit  := unsigned bits[1];

def M : matrix[8,8] of GF2 := { [1], [0], [0], [0], [1], [1], [1], [1],
                                [1], [1], [0], [0], [0], [1], [1], [1],
                                [1], [1], [1], [0], [0], [0], [1], [1],
                                [1], [1], [1], [1], [0], [0], [0], [1],
                                [1], [1], [1], [1], [1], [0], [0], [0],
                                [0], [1], [1], [1], [1], [1], [0], [0],
                                [0], [0], [1], [1], [1], [1], [1], [0],
                                [0], [0], [0], [1], [1], [1], [1], [1] };

def C : vector[8]   of GF2 := { [1], [1], [0], [0], [0], [1], [1], [0] };

def SBox( e : GF2N ) : GF2N {
  def x : GF2N;
  if (e == [0]) { x := [0]; }
  else { x := [1] / e; }
  def A : matrix[8,1] of GF2 := (matrix[8,1] of GF2)((GF2V)x);
  def B : GF2V := (GF2V)(M*A);
  return ((GF2N)B) + ((GF2N)C);
}

```

```
def SubBytes( s : S ) : S {
  def r : S;
  seq i := 0 to 3 { seq j := 0 to 3 { r[i,j] := SBox( s[i,j] );  } }
  return r;
}

def SubWord( w : vector[4] of GF2N ) : vector[4] of GF2N {
  def r : vector[4] of GF2N;
  seq i := 0 to 3 { r[i] := SBox( w[i] ); }
  return r;
}

def ShiftRows( s : S ) : S {
  def r : S;
  seq i := 0 to 3 { r[i,0..3] := (Row)(((RowV)s[i,0..3]) |> i); }
  return r;
}

def mix : matrix[4,4] of GF2N  := {
  [X],   [X+1], [1],   [1],
  [1],   [X],   [X+1], [1],
  [1],   [1],   [X],   [X+1],
  [X+1], [1],   [1],   [X]
};

def MixColumns( s : S ) : S {
  def r : S;
  seq i := 0 to 3 { r[0..3,i] := mix * s[0..3,i]; }
  return r;
}

def AddRoundKey( s : S, k : K ) : S {
  def r : S;
  seq i := 0 to 3 { seq j := 0 to 3 { r[i,j] := s[i,j] + k[i,j]; } }
  return r;
}

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

def makeRCon() : vector[11] of vector[4] of GF2N {
  def Rcon : vector[11] of vector[4] of GF2N;
  def lsw : vector[3] of GF2N := { [0],[0],[0] };

  seq i := 1 to 10 { def t : vector[1] of GF2N := { [X] };
    t[0]:=t[0]**(i 1); Rcon[i] := t @ lsw; }
  return Rcon;
}

def ExpandKey( k : K ) : vector[11] of K {
  def r : vector[11] of K;
  def Rcon : vector[11] of vector[4] of GF2N;
  Rcon := makeRCon();
  def lsw : vector[3] of GF2N := { [0], [0], [0] };

  r[0] := k;
  seq i := 1 to 10 { def oldr : K := r[i 1];
    def curr : K;
    curr[0..3,0] := ((Col)SubWord( ((ColV)oldr[0..3,3]) |> 1)) +
                                  ((Col)Rcon[i]) + oldr[0..3,0];
    seq j := 1 to 3 { curr[0..3,j] := curr[0..3,j 1] + oldr[0..3,j]; }
    r[i] := curr;
  }
  return r;
}

def Aes( s : S, k : K) : S {
  def r : S := s+k;
  def keys : vector[11] of K;
  keys := ExpandKey(k);
  seq i := 1 to 9 { r := FullRound( r,keys[i] ); }

  return ShiftRows( SubBytes(r) ) + keys[10];
}

def encodeByte( x : Byte ) : GF2N {
  def t : vector[8] of GF2 := { (GF2)x[0], (GF2)x[1], (GF2)x[2], (GF2)x[3],
                                (GF2)x[4], (GF2)x[5], (GF2)x[6], (GF2)x[7] };
  return (GF2N)t;
}

def encodeRow( x : vector[4] of Byte ) : vector[4] of GF2N {
  def t : vector[4] of GF2N;
  seq i := 0 to 3 { t[i] := encodeByte( x[i] ); }
  return t;
}

def encodeKey( inp : vector[16] of Byte ) : K {
  def r : K;
  seq i := 0 to 3 { seq j := 0 to 3 { r[i,j] := encodeByte( inp[i+4*j] ); } }
  return r;
}

def decodeByte( x : GF2N ) : Byte {
  def t : vector[8] of GF2;
  t := (vector[8] of GF2)x;
  def r : Byte := ((Bit)(int)t[0]) @ ((Bit)(int)t[1]) @
                  ((Bit)(int)t[2]) @ ((Bit)(int)t[3]) @
                  ((Bit)(int)t[4]) @ ((Bit)(int)t[5]) @
                  ((Bit)(int)t[6]) @ ((Bit)(int)t[7]);
  return r;
}

```

```
def decodeRow( x : matrix[1,4] of GF2N ) : vector[4] of Byte {
    def r : vector[4] of Byte;
    seq i := 0 to 3 { r[i] := decodeByte( x[0,i] ); }
    return r;
}

def decodeKey( x : K ) : vector[16] of Byte {
  def r : vector[16] of Byte;
  seq i := 0 to 3 { seq j := 0 to 3 { r[i*4+j] := decodeByte(x[j,i]); } }
  return r;
}

def fipstestK : vector[16] of Byte := {
  (Byte)0x2b, (Byte)0x7e, (Byte)0x15, (Byte)0x16,
  (Byte)0x28, (Byte)0xae, (Byte)0xd2, (Byte)0xa6,
  (Byte)0xab, (Byte)0xf7, (Byte)0x15, (Byte)0x88,
  (Byte)0x09, (Byte)0xcf, (Byte)0x4f, (Byte)0x3c
};

def fipstestS : vector[16] of Byte := {
  (Byte)0x32, (Byte)0x43, (Byte)0xf6, (Byte)0xa8,
  (Byte)0x88, (Byte)0x5a, (Byte)0x30, (Byte)0x8d,
  (Byte)0x31, (Byte)0x31, (Byte)0x98, (Byte)0xa2,
  (Byte)0xe0, (Byte)0x37, (Byte)0x07, (Byte)0x34
};

def fipsK : K := encodeKey(fipstestK);
def fipsS : S := encodeKey(fipstestS);
```

Bibliography

[1] Specification for the advanced encryption standard (aes), 2001. [cited at p. 43, 117]

[2] Onur Aciiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. On the power of simple branch prediction analysis. In *ACM symposium on Information, computer and communications security*, ASIACCS '07, pages 312–320. ACM, 2007. [cited at p. 87]

[3] Elvira Albert, Germán Puebla, and Manuel V. Hermenegildo. Abstraction-carrying code: a model for mobile code safety. *New Generation Comput.*, 26(2):171–204, 2008. [cited at p. 41]

[4] J. Bacelar Almeida, Manuel Barbosa, Jorge S. Pinto, and Bárbara Vieira. Deductive verification of cryptographic software. In Ewen Denny, editor, *Proceedings of the Second NASA Formal Methods Symposium (NFM 2009)*, Langley Research Center, Hampton VA 23681-2199, USA, April 2009. NASA. [cited at p. 6, 8]

[5] J. Bacelar Almeida, Manuel Barbosa, Jorge S. Pinto, and Bárbara Vieira. Formal verification of side-channel countermeasures using self-composition. *Science of Computer Programming*, 2011. [cited at p. 6, 8]

[6] José Bacelar Almeida, Manuel Barbosa, Jorge Sousa Pinto, and Bárbara Vieira. Verifying cryptographic software correctness with respect to reference implementations. In *FMICS'09*, volume 5825 of *LNCS*, pages 37–52, 2009. [cited at p. 6, 8]

[7] José Bacelar Almeida, Manuel Barbosa, Jorge Sousa Pinto, and Bárbara Vieira. Deductive verification of cryptographic software. *Innov. Syst. Softw. Eng.*, 6:203–218, September 2010. [cited at p. 6, 8, 61, 69, 72, 101]

[8] José Bacelar Almeida, Mario João. Frade, Jorge Sousa Pinto, and Simão Melo de Sousa. *Rigorous software development. An introduction to program verification.* London: Springer, 2011. [cited at p. 32]

[9] Manuel Barbosa, Andrew Moss, Dan Page, Nuno F. Rodrigues, and Paulo F. Silva. Type checking cryptography implementations, 2011. [cited at p. 110]

[10] Manuel Barbosa, Andrew Moss, Dan Page, Nuno F. Rodrigues, and Paulo F. Silva. Type checking cryptography implementations (full version). Technical Report DI-CCTC-11-01, CCTC, Univ. Minho, 2011. [cited at p. 108, 109]

[11] Manuel Barbosa, Jorge Sousa Pinto, Jean-Christophe Filliâtre, and Bárbara Vieira. A deductive verification platform for cryptographic software. *ECEASST*, 33, 2010. [cited at p. 7, 8]

[12] John Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003. [cited at p. 43]

[13] Mike Barnett, B. Evan Chang, Robert Deline, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Formal Methods for Components and Objects (FMCO 2005)*, volume 4111 of *LNCS*, pages 364–387. Springer-Verlag, 2006. [cited at p. 37]

[14] Mike Barnett, Peter Müller, Manuel Fähndrich, Wolfram Schulte, K. Rustan, M. Leino, and Herman Venter. Specification and verification: The spec # experience, 2009. [cited at p. 4, 32, 37]

[15] Mike Barnett, K. Rustan, M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In *CASSIS'04*, pages 49–69. Springer, 2004. [cited at p. 40]

[16] Gilles Barthe, Lilian Burdy, Julien Charles, Benjamin Grégoire, Marieke Huisman, Jean-Louis Lanet, Mariela Pavlova, and Antoine Requet. Jack - a tool for validation of security and behaviour of java applications. In Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever, editors, *FMCO*, volume 4709 of *Lecture Notes in Computer Science*, pages 152–174. Springer, 2006. [cited at p. 4, 38]

[17] Gilles Barthe, Pedro R. D'Argenio, and Tamara Rezk. Secure information flow by self-composition. In *CSFW*, pages 100–114. IEEE Computer Society, 2004. [cited at p. 6, 26, 28, 30, 39, 56, 83, 103]

[18] Gilles Barthe and César Kunz. An introduction to certificate translation. In Alessandro Aldini, Gilles Barthe, and Roberto Gorrieri, editors, *FOSAD*, volume 5705 of *Lecture Notes in Computer Science*, pages 51–95. Springer, 2008. [cited at p. 138, 144]

[19] Gilles Barthe and Tamara Rezk. A certified lightweight non-interference java bytecode verifier. In *European Symposium on Programming, Lecture Notes in Computer Science*. Springer, 2007. [cited at p. 30, 39]

[20] Gilles Barthe, Tamara Rezk, and Amitabh Basu. Security types preserving compilation. *Comput. Lang. Syst. Struct.*, 33(2):35–59, 2007. [cited at p. 30, 39]

[21] Patrick Baudin, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. *ACSL: ANSI/ISO C Specfication Language*. CEA LIST and INRIA, 2010. Version 1.5 (2009-2010). [cited at p. 4, 5, 32, 33]

[22] L. Bauer, J. Ligatti, and D. Walker. More enforceable security policies. In *In Foundations of Computer Security*, Copenhagen, Denmark, July 2002. [cited at p. 30]

[23] Lennart Beringer and Martin Hofmann. Secure information flow and program logics. In *CSF*, pages 233–248. IEEE Computer Society, 2007. [cited at p. 30, 41]

[24] Kenneth J. Biba. Integrity considerations for secure computer systems. In *USAF Electronic Systems Division, Air Force Systems Command*, April 1977. [cited at p. 38]

[25] Sascha Böhme, MichałMoskal, Wolfram Schulte, and Burkhart Wolff. Holboogie – an interactive prover-backend for the verifying c compiler. *J. Autom. Reason.*, 44(1-2):111–144, 2010. [cited at p. 4, 37]

[26] Dumitru Ceara, Laurent Mounier, and Marie-Laure Potet. Taint dependency sequences: A characterization of insecure execution paths based on input-sensitive cause sequences. In *ICSTW '10*, pages 371–380. IEEE, 2010. [cited at p. 45]

[27] Richard Chang, Guofei Jiang, Franjo Ivancic, Sriram Sankaranarayanan, and Vitaly Shmatikov. Inputs of coma: Static detection of denial-of-service vulnerabilities. In *CSF'09*, pages 186–199. IEEE, 2009. [cited at p. 45]

[28] Stephen Chong and Andrew C. Myers. Decentralized robustness. In *CSFW '06: Proceedings of the 19th IEEE workshop on Computer Security Foundations*, pages 242–256, Washington, DC, USA, 2006. IEEE Computer Society. [cited at p. 42]

[29] Lori A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Trans. Software Eng.*, 2(3):215–222, 1976. [cited at p. 31]

[30] Ernie Cohen, Markus Dahlweid, Mark A. Hillebrand, Dirk Leinenbach, Michal Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. Vcc: A practical system for verifying concurrent c. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *TPHOLs*, volume 5674 of *Lecture Notes in Computer Science*, pages 23–42. Springer, 2009. [cited at p. 4, 37]

[31] Common Criteria. Common criteria for information technology security evaluation, part 1: Introduction and general model. http://www.commoncriteriaportal.org/, September 2006. [cited at p. 4]

[32] Common Criteria. Common criteria for information technology security evaluation, part 2: Security functional requirements. http://www.commoncriteriaportal.org/, September 2006. [cited at p. 4]

[33] Common Criteria. Common criteria for information technology security evaluation, part 3: Security assurance requirements. http://www.commoncriteriaportal.org/, September 2006. [cited at p. 4]

[34] Sylvain Conchon, Evelyne Contejean, and Johannes Kanig. Ergo : a theorem prover for polymorphic first-order logic modulo theories, 2006. [cited at p. 5]

[35] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977. [cited at p. 31]

[36] Ádám Darvas, Reiner Hähnle, and David Sands. A theorem proving approach to analysis of secure information flow. In Dieter Hutter and Markus Ullmann, editors, *SPC*, volume 3450 of *Lecture Notes in Computer Science*, pages 193–209. Springer, 2005. [cited at p. 30]

[37] Leonardo de Moura and Nikolaj Bjørner. *Z3: An Efficient SMT Solver*, volume 4963/2008 of *Lecture Notes in Computer Science*, pages 337–340. Springer Berlin, April 2008. [cited at p. 37]

[38] Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Commun. ACM*, 20(7):504–513, 1977. [cited at p. 25, 38]

[39] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005. [cited at p. 5, 37]

[40] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice Hall, Inc., October 1976. [cited at p. 20, 35, 37]

[41] Edsger W. Dijkstra and Carel S. Scholten. *Predicate calculus and program semantics*. Texts and monographs in computer science. Springer, 1990. [cited at p. 40]

[42] Guillaume Dufay, Amy Felty, and Stan Matwin. Privacy-sensitive information flow with jml. In *Automated Deduction CADE-20*, pages 116–130. Springer Berlin / Heidelberg, August 2005. [cited at p. 30, 41]

[43] E. Allen Emerson and Edmund M. Clarke. Characterizing correctness properties of parallel programs using fixpoints. In J. W. de Bakker and Jan van Leeuwen, editors, *ICALP*, volume 85 of *Lecture Notes in Computer Science*, pages 169–181. Springer, 1980. [cited at p. 31]

[44] Jean-Christophe Filliâtre and Claude Marché. Multi-prover verification of c programs. In Jim Davies, Wolfram Schulte, and Michael Barnett, editors, *ICFEM*, volume 3308 of *Lecture Notes in Computer Science*, pages 15–29. Springer, 2004. [cited at p. 4, 33, 36, 37]

[45] Jean-Christophe Filliâtre and Claude Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In Werner Damm and Holger Hermanns, editors, *CAV*, volume 4590 of *Lecture Notes in Computer Science*, pages 173–177. Springer, 2007. [cited at p. 5, 34, 35, 111]

[46] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for java. In *ACM SIGPLAN Conference on Programming language design and implementation (PLDI'02)*, pages 234–245. ACM, 2002. [cited at p. 4, 33, 37]

[47] Robert W. Floyd. Assigning meanings to programs. In *Proc. Sympos. Appl. Math., Vol. XIX*, pages 19–32. Amer. Math. Soc., Providence, R.I., 1967. [cited at p. 4, 31, 39]

[48] Maria João Frade and Jorge Sousa Pinto. Verification conditions for source-level imperative programs. *Computer Science Review*, 5(3):252–277, 2011. [cited at p. 17, 23]

[49] Roberto Giacobazzi and Isabella Mastroeni. Abstract non-interference: parameterizing non-interference by abstract interpretation. *SIGPLAN Not.*, 39(1):186–197, 2004. [cited at p. 41, 42]

[50] J.A. Goguen and J. Meseguer. Security policies and security models. *In Proc. IEEE Symp. on Security and Privacy*, pages 11–20, 1982. [cited at p. 25]

[51] Gurvan Le Guernic, Anindya Banerjee, Thomas Jensen, and David A. Automata-based confidentiality monitoring. In *In Proceedings of 11th Annual Asian Computing Science Conference (ASIAN 2006)*, 2006. [cited at p. 30]

[52] Paolo Herms. Certification of a chain for deductive program verification. In Yves Bertot, editor, *2nd Coq Workshop, satellite of ITP'10*, Edinburgh, Royaume-Uni, 2010. [cited at p. 138, 144]

[53] Paolo Herms, Claude Marché, and Benjamin Monate. A Certified Multi-prover Verification Condition Generator. Research Report RR-7793, INRIA, 2011. [cited at p. 138]

[54] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969. [cited at p. 4, 14, 31, 39]

[55] Peter Homeier and David Martin. Trustworthy tools for trustworthy programs: A verified verification condition generator. In Thomas Melham and Juanito Camilleri, editors, *Higher Order Logic Theorem Proving and Its Applications*, volume 859 of *Lecture Notes in Computer Science*, pages 269–284. Springer Berlin / Heidelberg, 1994. [cited at p. 138, 144]

[56] B. P. F. Jacobs, J. R. Kiniry, M. E. Warnier, Bart Jacobs, Joseph Kiniry, and Martijn Warnier. Java program verification challenges. In *FMCO'02*, volume 2852 of *LNCS*, pages 202–219. Springer, 2003. [cited at p. 40]

[57] Rajeev Joshi and K. Rustan M. Leino. A semantic approach to secure information flow. *Sci. Comput. Program.*, 37:113–138, May 2000. [cited at p. 30, 39]

[58] Emilia Käsper and Peter Schwabe. Faster and timing-attack resistant aes-gcm. In Christophe Clavier and Kris Gaj, editors, *CHES*, volume 5747 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 2009. [cited at p. 87]

[59] Dexter Kozen. Language-based security. In Miroslaw Kutylowski, Leszek Pacholski, and Tomasz Wierzbicki, editors, *MFCS*, volume 1672 of *Lecture Notes in Computer Science*, pages 284–298. Springer, 1999. [cited at p. 29, 30]

[60] Len Lapadula, Leonard J. Lapadula, and D. Elliott Bell. Secure computer systems: Mathematical foundations, 1973. [cited at p. 38]

[61] Gurvan Le Guernic. Precise Dynamic Verification of Noninterference. Technical report, INRIA-MSR, 2008. [cited at p. 30]

[62] Gary T. Leavens, Clyde Ruby, K. Rustan, M. Leino, Erik Poll, and Bart Jacobs. Jml (poster session): notations and tools supporting detailed design in java. In *OOPSLA '00: Addendum to the 2000 proceedings of the conference on Object-oriented programming, systems, languages, and applications (Addendum)*, pages 105–106, New York, NY, USA, 2000. ACM. [cited at p. 32, 33, 37, 40]

[63] Peng Li and Steve Zdancewic. Downgrading policies and relaxed noninterference. In Jens Palsberg and Martín Abadi, editors, *POPL*, pages 158–170. ACM, 2005. [cited at p. 40, 42]

[64] Heiko Mantel and David Sands. Controlled declassification based on intransitive noninterference. In *Programming Languages and Systems*, volume Volume 3302/2004, pages 129–145. Springer Berlin / Heidelberg, 2004. [cited at p. 41, 42]

[65] Manuel Barbosa (editor). CACE Deliverable 5.1: Security Policies for Cryptographic Software, January 2009. [cited at p. 5]

[66] Manuel Barbosa (editor). CACE Deliverable 5.2: Machine assisted verification and certification tools, June 2010. [cited at p. 107, 112]

[67] C. Marché, C. Paulin-mohring, and X. Urbain. The krakatoa tool for certication of java/javacard programs annotated in jml, 2004. [cited at p. 36, 41, 111]

[68] David Molnar, Matt Piotrowski, David Schultz, and David Wagner. The program counter security model: Automatic detection and removal of control-flow side channel attacks. In *Proceedings of ICISC'05*, volume 3935 of *LNCS*, pages 156–168. Springer, 2006. [cited at p. 44, 87]

[69] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From system f to typed assembly language. *ACM Trans. Program. Lang. Syst.*, 21(3):527–568, 1999. [cited at p. 30]

[70] Yannick Moy. *Automatic Modular Static Safety Checking for C Programs*. PhD thesis, Université Paris-Sud, January 2009. [cited at p. 36]

[71] Andrew C. Myers. Jflow: Practical mostly-static information flow control. In *In Proc. 26th ACM Symp. on Principles of Programming Languages (POPL*, pages 228–241, 1999. [cited at p. 42]

[72] Andrew C. Myers and Barbara Liskov. A decentralized model for information flow control. In *Symposium on Operating Systems Principles*, pages 129–142, 1997. [cited at p. 30, 41, 42]

[73] Andrew C. Myers and Barbara Liskov. Complete, safe information flow with decentralized labels. In *In Proc. IEEE Symposium on Security and Privacy*, pages 186–197, 1998. [cited at p. 30, 41, 42]

[74] Andrew C. Myers, Barbara Liskov, and Name Barbara Liskov. Protecting privacy using the decentralized label model. *ACM Transactions on Software Engineering and Methodology*, 9:2000, 2000. [cited at p. 30, 41, 42]

[75] Andrew C. Myers, Andrei Sabelfeld, and Steve Zdancewic. Enforcing robust declassification and qualified robustness. *Journal of Computer Security*, 2006:2006, 2006. [cited at p. 41, 42]

[76] David A. Naumann. From coupling relations to mated invariants for checking information flow. In Dieter Gollmann, Jan Meier, and Andrei Sabelfeld, editors, *ESORICS*, volume 4189 of *Lecture Notes in Computer Science*, pages 279–296. Springer, 2006. [cited at p. 30, 40]

[77] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of aes. In *Topics in Cryptology - CT-RSA 2006, The Cryptographers' Track at the RSA Conference 2006*, pages 1–20. Springer-Verlag, 2005. [cited at p. 87]

[78] S. Owre, J. M. Rushby, , and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, jun 1992. Springer-Verlag. [cited at p. 43]

[79] Dan Page. Detailed cao and qhasm language specifications. Technical Report Deliverable D1.1, CACE Project, 2009. [cited at p. 108]

[80] A. Sabelfeld and A. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1), 2003. [cited at p. 39]

[81] Andrei Sabelfeld and David Sands. Declassification: Dimensions and principles. *J. Computer Security*, 2007. [cited at p. 41, 42]

[82] Fred B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1):30–50, 2000. [cited at p. 30]

[83] Fred B. Schneider, J. Gregory Morrisett, and Robert Harper. A language-based approach to security. In Reinhard Wilhelm, editor, *Informatics*, volume 2000 of *Lecture Notes in Computer Science*, pages 86–101. Springer, 2001. [cited at p. 29, 30]

[84] Bruce Schneier. *Applied cryptography: protocols, algorithms, and source code in C*. Wiley, New York, 2nd edition, 1996. [cited at p. 51]

[85] Martijn Stam. On montgomery-like representationsfor elliptic curves over gf(2^k). In Yvo Desmedt, editor, *Public Key Cryptography*, volume 2567 of *Lecture Notes in Computer Science*, pages 240–253. Springer, 2003. [cited at p. 131]

[86] Elisabeth A. Strunk, Xiang Yin, and John C. Knight. Echo: a practical approach to formal verification. In *Proceedings of the 10th international workshop on Formal methods for industrial critical systems*, FMICS '05, pages 44–53, New York, NY, USA, 2005. ACM. [cited at p. 43]

[87] Josef Svenningsson and David Sands. Specification and verification of side channel declassification. In *FAST'09*, volume 5983 of *LNCS*, pages 111–125. Springer, 2009. [cited at p. 44, 45, 87, 88, 100, 101, 102]

[88] Tachio Terauchi and Alexander Aiken. Secure information flow as a safety problem. In Chris Hankin and Igor Siveroni, editors, *SAS*, volume 3672 of *Lecture Notes in Computer Science*, pages 352–367. Springer, 2005. [cited at p. 30, 40, 41, 56, 88, 101, 148]

[89] The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version V8.2*, 2008. [cited at p. 5]

[90] Stephen Tse and Steve Zdancewic. A design for a security-typed language with certificate-based declassification. In *In Proc. European Symp. on Programming, volume 3444 of LNCS*, pages 279–294. Springer-Verlag, 2005. [cited at p. 42]

[91] Jeffrey A. Vaughan and Steve Zdancewic. A cryptographic decentralized label model. In *SP '07: Proceedings of the 2007 IEEE Symposium on Security and Privacy*, pages 192–206, Washington, DC, USA, 2007. IEEE Computer Society. [cited at p. 41, 42]

[92] Frédéric Vogels, Bart Jacobs, and Frank Piessens. A machine-checked soundness proof for an efficient verification condition generator. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, pages 2517–2522, New York, NY, USA, 2010. ACM. [cited at p. 138, 144]

[93] Dennis M. Volpano, Cynthia E. Irvine, and Geoffrey Smith. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2/3):167–188, 1996. [cited at p. 30, 38]

[94] Dennis M. Volpano and Geoffrey Smith. Eliminating covert flows with minimum typings. In *CSFW*, pages 156–169. IEEE Computer Society, 1997. [cited at p. 30, 44]

[95] Martijn Warnier and Martijn Oostdijk. Non-interference in JML. Technical Report ICIS-R05034, Nijmegen Institute for Computing and Information Sciences, 2005. [cited at p. 30, 40]

[96] Xiang Yin, J. Knight, and W. Weimer. Exploiting refactoring in formal verification. In *Dependable Systems Networks, 2009. DSN '09. IEEE/IFIP International Conference on*, pages 53 –62, 29 2009-july 2 2009. [cited at p. 43]

[97] Xiang Yin, John C. Knight, Elisabeth A. Nguyen, and Westley Weimer. Formal verification by reverse synthesis. In *Proceedings of the 27th international conference on Computer Safety, Reliability, and Security*, SAFECOMP '08, pages 305–319, Berlin, Heidelberg, 2008. Springer-Verlag. [cited at p. 43]

[98] Dachuan Yu and Islam Nayeem. A typed assembly language for confidentiality. *Lecture notes in computer science*, 3924:162–179, 2006. [cited at p. 30, 39]

[99] Steve Zdancewic. A type system for robust declassification, 2004. [cited at p. 42]

[100] Steve Zdancewic and Andrew C. Myers. Robust declassification. In *in Proc. IEEE Computer Security Foundations Workshop*, pages 15–23. IEEE Computer Society Press, 2001. [cited at p. 42]

[101] Lantian Zheng and Andrew C. Myers. End-to-end availability policies and noninterference. In *CSFW '05: Proceedings of the 18th IEEE workshop on Computer Security Foundations*, pages 272–286, Washington, DC, USA, 2005. IEEE Computer Society. [cited at p. 42]

[102] Lantian Zheng and Andrew C. Myers. Dynamic security labels and static information flow control. *Int. J. Inf. Secur.*, 6(2):67–84, 2007. [cited at p. 42]