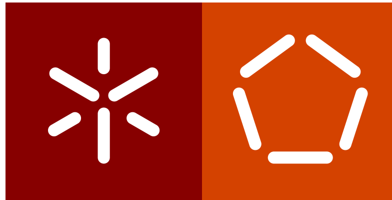
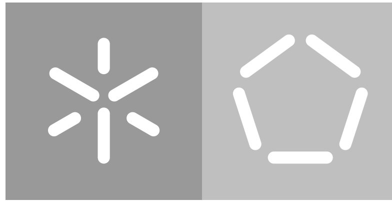


Universidade do Minho
Escola de Engenharia
Departamento de Informática

Luís Henrique Martins

Geração de descrições de computação para a *cloud*

March 2017



Universidade do Minho
Escola de Engenharia
Departamento de Informática

Luís Henrique Martins

Geração de descrições de computação para a *cloud*

Master dissertation
Master Degree in Computer Science

Dissertation supervised by
José Bacelar Almeida
Manuel Bernardo Barbosa

March 2017

AGRADECIMENTOS

Antes de mais gostaria de agradecer aos meus orientadores, Professor José Bacelar Almeida e Professor Manuel Bernardo Barbosa, pela colaboração e disponibilidade, que permitiram a realização desta dissertação.

Agradeço também aos meus amigos e colegas de trabalho que me acompanharam não só durante o desenvolvimento desta dissertação de mestrado, mas também por todo o percurso académico.

Por fim, agradeço aos meus pais, à minha irmã e ao meu irmão pelo apoio incondicional durante todos estes anos e que me permitiram chegar hoje onde cheguei.

This work was supported by the European Union Seventh Framework Program (FP7/2007-2013) under grant agreement n. 609611 (PRACTICE).

ABSTRACT

Over the last years, the demand of a secure computation in the cloud has been a growing concept in which people are taking interest in.

The attractiveness of his theme has been driving the arise of protocols proposals that take advantages in cloud computing characteristics. However, to solve this need the majority of these solutions require that their features to be implemented in a very low-level of abstraction, more precisely in the format of logical Boolean circuits.

Clearly it is not simple, neither work productive, to implement these specifications in such a low-level of abstraction. So there is a need to transform the description of the functionality implemented in a higher level language into those circuits. This project is based on the study of this transformation, ensuring its correction and the semantic preservation of the source code.

In order to perform this transformation we propose a certified compiler which will be able to generate descriptions of Boolean circuits from a C programs with certain constraints. It will be also take into account the degree of eficiency of these descriptions, keeping its correctness.

RESUMO

Nos últimos anos, a procura por soluções que tirem partido de um computação segura na *cloud* é um conceito em expansão e de grande interesse. A atratividade deste tema tem motivado a apresentação de inúmeras propostas de protocolos que tiram partido dessas características. Contudo, a grande maioria desses protocolos requerem que as funcionalidades a executar se apresentem descritos a um nível de abstração muito baixo, concretamente na forma de circuitos lógicos Booleanos.

Obviamente que não é simples nem produtivo trabalhar a esse nível de abstração, pelo que surge uma necessidade de converter descrições de programas realizado numa linguagem de alto-nível nesses circuitos. Este projeto baseia-se no estudo dessa transformação, assegurando que a mesma é correta garantindo a preservação da semântica do código fonte.

Para a realização desta transformação será proposto um compilador certificado, que terá a intenção de gerar descrições de circuitos Booleanos a partir de programas C. Para a produção destas descrições será tido em conta a sua eficiência de forma a melhorar a sua performance mantendo a fiabilidade do mesmo.

CONTEÚDO

1 INTRODUÇÃO 1
- 1.1 Contextualização 1
  - 1.1.1 Projeto Europeu - PRACTICE 1
  - 1.1.2 Compilador Certificado - CompCert C 2
  - 1.1.3 Circuitos Lógicos Booleanos 3
- 1.2 Motivação 3
- 1.3 Objetivos do trabalho 4
- 1.4 Estrutura do documento 5

2 ESTADO DA ARTE 6
- 2.1 Compilação certificada 6
  - 2.1.1 Abordagens de verificação de compiladores 6
  - 2.1.2 Preservação semântica 7
  - 2.1.3 Assistente de prova Coq 8
- 2.2 O caso do CompCert C 9
- 2.3 Circuitos Booleanos 12
  - 2.3.1 Especificação de circuitos Booleanos 12
  - 2.3.2 Circuitos de garbled 14
  - 2.3.3 Otimizações de circuitos 15
- 2.4 CBMC-GC 16
  - 2.4.1 CBMC 16
  - 2.4.2 Ferramenta CBMC-CG 17
- 2.5 Conclusões 18

3 CIRCUIT DESCRIPTION GENERATION (CDG) 19
- 3.1 Arquitetura do compilador 19
  - 3.1.1 Restrições do compilador 20
- 3.2 Front-End 21
  - 3.2.1 Validação Estrutural 21
- 3.3 Back-End 22
  - 3.3.1 Register Transfer Language Circuits (RTLC) 22
  - 3.3.2 High Level Circuit (HLC) 23
  - 3.3.3 Low Level Circuits 25

4 CONTRIBUIÇÕES 28
- 4.1 Oráculo de previsão de nível de *unrolling* de ciclos 28

4.1.1 Formato da linguagem ambiente 29
4.1.2 Processo de realização do oráculo 30
4.2 Ferramenta geradora de circuitos 32
4.2.1 Low Level Boolean Circuit 33
4.2.2 High Level Circuit file description 33
4.2.3 Funcionamento da ferramenta 35
4.2.4 Funcionalidades adicionais e modo de utilização 39
4.3 Funcionalidades a suportar 40
5 ANÁLISE DO CDG 42
5.1 Análise comparativa 42
5.2 Caso de estudo 44
6 CONCLUSÕES E TRABALHO FUTURO 49
6.1 Conclusões 49
6.2 Trabalho Futuro 50

LISTA DE FIGURAS

Figura 1	Estrutura global do CompCert C	10
Figura 2	Esquema de um circuito - fulladder	13
Figura 3	Arquitetura geral da ferramenta CBMC-GC	17
Figura 4	Arquitetura do compilador certificado de circuitos	20
Figura 5	Arquitetura da ferramenta geradora de circuitos de baixo nível	32
Figura 6	Comparação entre o CBMC-CG e o CDG para diferentes algoritmos	43
Figura 7	Comparação entre os circuitos de Bristol e o CDG para diferentes algoritmos	44
Figura 8	Comparação do número dos vários tipos *gate* para os diversos resultados do AES	45
Figura 9	Comparação do número dos vários tipos *gate* entre os dois resultados do AES	46
Figura 10	Comparação do número dos vários tipos *gate* para os diversos resultados do SHA256	47

LISTA DE TABELAS

Tabela 2 Opções de linha de comando da ferramenta com descrição da funcionalidade e respetivo argumento 40
Tabela 3 Número de cada tipo de *gate* de seleção nas três implementações do Advanced Encryption Standard (AES) 46

LISTA DE ACRÓNIMOS

AES — Advanced Encryption Standard.
AST — Árvores de Sintaxe Abstracta.

CBMC — C-Bounded Model Checker.
CDG — Circuit Description Generation.
CIC — Calculus of Inductive Constructions.
CLI — Command Language Interpreter.

FHE — Fully Homomorphic Encryption.

HLC — High Level Circuit.

MEI — Mestrado em Engenharia Informática.
MPC — Secure Multiparty Computation.

OCaml — Objective Caml.

RTL — Register Transfer Language.
RTLC — Register Transfer Language Circuits.

SFE — Two-party secure function evaluation.
SHA256 — Secure Hash Algorithm - 256.
SSA — Single Static Assigment.
STC — Secure Two-Party Computation.

UM — Universidade do Minho.

1

INTRODUÇÃO

1.1 CONTEXTUALIZAÇÃO

Para a compreensão de todo o contexto geral da dissertação é fundamental fazer referência aos projetos em que esta se insere, com o objetivo de perceber o motivo pelo qual faz sentido tirar partido das características de ambos os projetos. Ainda em forma de contextualização irá ser introduzido o conceito de circuito Booleano, importante também para a compreensão dos objetivos da dissertação.

1.1.1 *Projeto Europeu - PRACTICE*

Atualmente, quando se discute a utilização da *cloud*, surge recorrentemente o problema da proteção de dados. Dadas as suas características, a *cloud* é um serviço que fornece benefícios, não só a nível financeiro, como também a níveis de usabilidade e de acessibilidade. Assim, é com naturalidade que organizações e indivíduos optam por mover os seus dados para a *cloud*.

Requisitos como confidencialidade e integridade são atingidos com relativa facilidade quando falamos de uma infraestrutura local, contudo tornam-se complicados de atingir quando existe a interação de um serviço externo.

O objetivo do projeto PRACTICE [1]é construir tecnologias que permitam computações seguras na *cloud*. De forma a manter os dados usados entre processos na *cloud* secretos, o projeto PRACTICE irá permitir que quer os fornecedores do serviço, quer outras entidades não autorizadas, não consigam obter informação secreta ou sensível.

Torna-se, então, importante fornecer ao utilizador do serviço confiança nas medidas de segurança da informação desenvolvidas. Estas medidas passam pela utilização de mecanismos criptográficos que permitam a computação em dados cifrados. Assim, é objetivo do PRACTICE endereçar diferentes situações onde esta computação é necessária, como a proteção dos dados do utilizador para com outros utilizadores do serviço e também do

1 http://www.practice-project.eu/

fornecedor de serviço, e permitir uma computação segura entre servidores e agentes não confiáveis.

O projeto pretende desenvolver diversas tecnologias fundamentais e depois, construi-las distintamente mas com desenvolvimentos complementares. As principais tecnologias que pretende investigar são:

- Secure Multiparty Computation (MPC)
- Fully Homomorphic Encryption (FHE)
- Ferramentas de desenvolvimento de domínio especifico
- Aplicação de métodos formais para verificar propriedades relevantes dos sistemas resultantes

De acordo com os princípios deste projeto estas tecnologias serão investigadas de forma a desenvolve-las de forma simultânea. O projeto pretende o desenvolvimento de novas linguagens de programação e de novas ferramentas para suportar aplicações que combinem o uso de tecnologias como MPC ou FHE.

1.1.2 *Compilador Certificado - CompCert C*

A ideia da criação de um compilador certificado não é nova. A missão de usar o computador para verificar provas de que os compiladores são corretos foi utilizada por McCarthy and Painter [1967], na qual foi feita a prova da correção de um simples algoritmo para compilar expressões aritméticas em linguagem máquina. Outro exemplo desta ideia foi abordada por Milner and Weyrauch [1972] com o intuito de provar a correção de uma máquina que transformava uma linguagem simples(ALGOL) numa linguagem *assembly*.

O projeto *CompCert* [2]tem como principal resultado um compilador certificado (*CompCert C*), altamente confiável para um subconjunto da linguagem C (ISO C90 / ANSI C standard), gerando código verificado para processadores PowerPC, ARM e x86.

Usando o *Coq proof assistant* para programar e provar a correção do programa, é garantido que o comportamento do código fonte é preservado no código compilado. Esta preservação assegura que não há a possibilidade de introdução de bugs por parte do compilador, permitindo o uso deste em contexto de software critico. Para software não critico, a má compilação é um problema mas não é uma questão fundamental, pois a ocorrência de bugs é pouco provável e pode ser ignorada. Contudo, quando vidas humanas, infraestruturas importantes ou informação sensível estão em risco, não pode existir a possibilidade de erro, logo uma falha na compilação é um problema que não pode ser ignorado.

2 http://compcert.inria.fr/

Segundo o estudo realizado por Eide and Regehr [2008] estas falhas na compilação não são um problema leviano, que raramente ou simplesmente não ocorrem. As falhas na compilação acontecem, e em contextos em que existe um pressuposto de confiança nos compiladores.

> *We tested thirteen production-quality C compilers and, for each, found situations in which the compiler generated incorrect code for accessing volatile variables. This result is disturbing because it implies that embedded software and operating systems — both typically coded in C, both being bases for many mission-critical and safety-critical applications, and both relying on the correct translation of volatiles — may be being miscompiled.* Eide and Regehr

1.1.3 *Circuitos Lógicos Booleanos*

Nos anos 1850, George Boole desenvolveu um método para computar conclusões lógicas. A sua lógica Booleana é uma teoria matemática na qual todas as variáveis podem apenas ter dois valores (verdadeiro ou falso). A partir desta teoria simples, foram desenvolvidas funções booleanas e criados sistemas binários que hoje são utilizados em circuitos elétricos e em computadores. Nas computações destes sistemas estão *gates* que permitem a computação de múltiplas variáveis e a agregação desses *gates* possibilita uma mistura de computações que poderão perfazer tanto algoritmos mais simples como os mais complexos.

Devido à sua simplicidade e universalidade (isto é, a capacidade de descrever qualquer computação), estes circuitos são bastante utilizados em determinados contextos da criptografia. Cifras homomórficas e protocolos de MPC são exemplos de técnicas que tiram partido destas características para avaliar computações aritméticas. É claro que utilizar uma técnica com este tipo de caraterísticas torna-se necessário arranjar uma forma de implementar a funcionalidade pretendida como um circuito booleano, ou seja sem ciclos, nem condições, entre outros, usando apenas operações lógicas simples. Todas estas restrições trazem um óbvio aumento da complexidade do código, e, por consequência, uma maior dificuldade na implementação.

1.2 MOTIVAÇÃO

Um problema de contexto critico

Uma parte desta dissertação tem como objetivo o estudo do compilador certificado CompCert C, surgindo algumas questões quando se questiona a qualidade de um compilador. O principal propósito de um compilador é o de transformar código fonte escrito numa determinada linguagem noutro: código objeto. Nesta transformação é esperado que a semântica

do código compilado se comporte de acordo com o elaborado no programa fonte. Contudo, isto nem sempre é garantido. Grande parte dos compiladores atuais permitem a ocorrência de bugs que podem provocar *crashs* no momento da compilação, ou mesmo a geração de um executável incorreto a partir de um programa origem correto, sendo a deteção e a correção destes erros bastante complicadas.

Em contextos de software critico a não ocorrência destes erros é essencial. Para garantir a sua inexistência é necessária a utilização de métodos formais em detrimento da validação por testes. Assim, a partir momento em que estes métodos são aplicados sistematicamente no código fonte, o compilador pode ser o elo mais fraco nesta cadeia, desde a especificação até aos executáveis. Isto porque a ocorrência de bugs provocados pelo compilador pode invalidar todas as garantias asseguradas pela utilização de métodos formais.

Apesar de existirem algumas técnicas para reduzir os problemas apontados, como revisões manuais do código *assembly* gerado, os bugs podem prejudicar o tempo de desenvolvimento bem como a performance geral do programa. Surge, assim, o *CompCert C*, um resultado da aplicação de métodos formais ao compilador, que assegura a preservação da semântica do código ao longo do processo de compilação.

Cenário de utilização

A existência de um compilador certificado permite que, numa determinada fase do seu processo de compilação, se consiga transformar descrições de uma linguagem de alto nível para descrições de funcionalidades expressas em circuitos booleanos. Esta garantia de transformação correta e certificada é bastante útil quando se pretende utilizar estes circuitos em protocolos de computação segura.

Tal como referido anteriormente, o projeto PRACTICE zela por alcançar técnicas que permitam uma computação segura na *cloud*. Para alcançar este fim, a utilização de circuitos booleanos é bastante comum em técnicas criptográficas. Assim, faz todo o sentido tirar partido das garantias de um compilador certificado, permitindo a utilização do mesmo no PRACTICE conseguindo, deste modo, atingir os principais objetivos desta dissertação.

1.3 OBJETIVOS DO TRABALHO

Esta investigação está inserida no projeto europeu PRACTICE e tem como principal objetivo a implementação de um compilador certificado de circuitos, que transforma programas numa linguagem C em especificações de circuitos Booleanos. Para a realização do mesmo será usado como base o compilador certificado CompCert com o intuito de tirar partido das suas características formais.

Inicialmente define-se como meta o estudo deste compilador certificado de forma a perceber o seu funcionamento geral, bem como os métodos de compilação certificada usados. Este estudo do cenário é fundamental para conseguir tirar benefícios da formalização da linguagem do CompCert bem como para uma transformação correta de uma linguagem de alto nível para circuitos booleanos.

Após a compreensão de todo o contexto e conceitos envolventes ao CompCert, será o desenvolvimento da ferramenta de conversão de descrições. Esta fase envolve a adaptação ao ambiente do CompCert e a realização da componente responsável pela conversão de entre descrições. Por último também é objetivo a análise de resultados obtidos pelo compilador e compara-los com outras descrições alternativas de ferramentas alternativas, como o CBMC-CG.

1.4 ESTRUTURA DO DOCUMENTO

Este documento encontra-se dividido em seis capítulos. Neste capítulo inicial é introduzido o tema da dissertação, sendo enunciado os contextos nos quais este projeto se insere e é explicada as principais motivações que movem a realização desta investigação.

No capítulo 2 é detalhado o estado da arte relevante para o âmbito desta dissertação. É introduzido o conceito de compilação certificada e explorados os conceitos inerentes ao mesmo e é explorado o exemplo do CompCert C. Posteriormente é formalizada a definição de circuito Booleano e é referido e explicada um ferramenta importante no contexto do projeto o CBMC-CG.

No capítulo 3 é apresentado o compilador certificado de circuitos que foi elaborado pelo grupo de investigação no qual este projeto se insere. São detalhadas as diferentes fases do compilador, sendo referidos os aspetos mais relevantes de cada uma.

No capítulo 4 são referidas e explicadas as contribuições realizadas para o compilador certificado. O trabalho desenvolvido poderá ser visto em duas componentes distintas: um oráculo de previsão do nível de *unrolling* de um ciclo e uma ferramenta de transformação de descrições de alto nível para outras num nível de abstração mais baixo.

No capítulo 5 são apresentados os resultados das descrições geradas pelo compilador que são comparados com resultados de outras descrições para um mesmo algoritmo. Ainda é realizado um caso de estudo de forma a evidenciar determinados aspetos a expor.

Por fim, no capítulo 6 é realizada uma síntese do trabalho efetuado e dos resultados obtidos. É também realizada uma reflexão sobre o trabalho futuro que poderia ser elaborado de forma a melhorar o compilador.

2

ESTADO DA ARTE

Nesta secção irão ser abordadas as principais noções para a realização de um compilador certificado, bem como as formas de o provar. De seguida será explicado o caso do especifico do CompCert que utiliza essas técnicas de compilação certificada para a sua construção e prova, detalhando as diversas fases que o compõem. Por fim, também irá ser abordado o conceito de circuitos booleanos útil para o objetivo final do projeto. Também serão abordadas algumas ferramentas que têm objetivos similares ao deste projeto, mas que contudo foram abordadas de outro prisma.

2.1 COMPILAÇÃO CERTIFICADA

Antes de explicar certas noções de compilação certificada serão necessárias algumas considerações sobre a notação definida. S simboliza um programa fonte arbitrário e C o código compilado. B designa o comportamento observável através da execução de um programa. A representação $P \Downarrow B$ significa que o programa P executa de acordo com o comportamento observacional B e a relação $S \approx C$ significa que os programa S e C preservam as propriedades da preservação de semântica.

2.1.1 *Abordagens de verificação de compiladores*

Para garantir uma compilação certificada é necessário a utilização de métodos corretos e eficazes capazes de executar tal fim. Assim, é importante referir as alternativas de como realizar a prova da conservação da semântica. Existem três possíveis tipos de abordagem referidas por Leroy [2009a], cada uma com características peculiares que são utilizadas dependendo do contexto da prova.

Compiladores Verificados. *Assume-se o compilador é uma função total (Comp) que ou gera o programa compilado (Comp(S) = OK(C)) ou reporta um erro na compilação(Comp(S) = Erro). Isto*

significa que é necessário aplicar tecnologias de prova ao programa fonte do compilador. Assim, o compilador encontra-se verificado se provar a seguinte propriedade:

$$\forall S, C, Comp(S) = OK(C) \Rightarrow S \approx C \tag{1}$$

Translation validation. *Nesta abordagem não é necessário a verificação do compilador. Pelo contrario, é feita a verificação a posteriori da propriedade C $\approx$ S através de uma função booleana de validação. Assim se Comp(S)=OK(C) e Validate(S,C) = true, pode-se concluir que o código compilado é correto. Contudo, a utilização desta função implica que a mesma esteja correta, por isso é necessário que seja acompanhada por uma prova formal da sua verificação. Assim, pode-se afirmar que a combinação entre essa função e um compilador não verificado traz as mesma garantias de correção de um compilador verificado. Esta abordagem é mais apetecível que a anterior pois apenas é necessária fazer a prova da função booleana que é significativamente mais pequena que a do compilador.*

Proof-carrying code e compiladores certificados. *Esta alternativa utiliza um compilador certificado que produz o código compilado juntamente com uma prova de que este código satisfaz determinadas especificações comportamentais. Tal como a abordagem anterior não é necessário verificar o compilador nem confiar na entidade que produz o código, apenas é precisa tornar confiável o programa que verifica a prova especificada anteriormente.*

2.1.2 *Preservação semântica*

Como já foi referido anteriormente, num processo de compilação certificada é imperativo que o código executável produzido pelo compilador se comporta exatamente como foi especificado pela semântica do código fonte, impossibilitando qualquer risco de má compilação. Segundo Leroy, a preservação semântica pode ser defina como:

Preservação Semântica. *Para todos os programas fonte S e todo o código C gerado pelo compilador, se o compilador, aplicando a fonte S produz código C, sem reportar nenhum erro de compilação então o comportamento observável executando de C é um dos possíveis comportamentos observável executando S.*

Complementando esta definição, é permitido ao compilador falhar a compilação, não gerando código. Isto pode acontecer se S é sintaticamente incorreto, se origina um erro de execução ou se a capacidade interna do compilador é excedida. Caso não falhe a compilação, é implementado um dos comportamentos possíveis do programa fonte. De forma a tornar precisa a prova da compilação foi introduzida a noção de comportamento observacional. Estes podem ser classificados como de terminação, divergência ou de erro e são definidos como:

Comportamento Observacional. *Engloba tudo aquilo que o utilizador ou o ambiente em que é executado pode ver acerca das ações do programa, perfazendo um traço de todas as operações I/O realizadas juntamente com a indicação se e como o programa terminou.*

Estes comportamentos refletem de forma precisa aquilo que o utilizador do programa ou a entidade que esteja a interagir com o programa pode observar. Com estas definições é possível deduzir diversas noções de preservação semântica. Algumas mais forte que outras, e portanto mais complicadas de realizar a sua prova. A noção de preservação semântica é na realidade mais subtil, e é necessário a consideração de diversos cenários quando se utiliza determinadas noções da mesma, considere-se estes dois exemplos.

Em determinados casos, se a avaliação de S resulta em Erro então o mesmo também acontece a C. Esta noção é demasiado forte porque pode impossibilita a realização de certas otimizações por parte do compilador (existência de uma operação invalida em S, mas que após a compilação é eliminada).

Outro caso a ter conta é quando ocorre um comportamento não especificado em S, ou seja se a linguagem fonte é não determinística e C pode ter comportamentos não desejados, para além dos especificados em S. Isto não ocorre quando a linguagem fonte é determinística, obrigando à admissão de apenas um comportamento observacional em C.

Tendo em conta estas duas condições podemos definir uma noção possível (menos informativa e com uma prova mais fácil):

$$\forall B \notin Erro, S \Downarrow B \Rightarrow C \Downarrow B \tag{2}$$

Esta abordagem para caraterizar uma preservação semântica entre dois programas ($C \approx S$) é a utilizada pelo compilador certificado na abordagem às provas de teorias.

2.1.3 *Assistente de prova Coq*

No caso em estudo, CompCert, é utilizado o Coq para realizar a verificação formal do compilador. *Coq proof assistant* 2009 é uma ferramenta de software que ajuda a construir a prova através da interação com o utilizador, verificando a validade da prova. Tal mecanismo de prova fornece um nível de confiança bastante elevado sobre a sua validade. O Coq foi desenvolvido para realizar provas matemáticas e escrever especificações formais, programas e provas de que programas executam de acordo com a sua especificação. Propriedades, programas e provas são formalizadas numa linguagem chamada Calculus of Inductive Constructions (CIC).

Este pode ser utilizado não só para prova da correção, mas também para programar grande parte do compilador. Esta é umas das funcionalidades importantes deste software

pois possibilita a extração de programas executáveis a partir de especificações, sendo o código fonte Objective Caml (OCaml)(apresentado por Leroy et al. [2002]) ou Haskell.

A utilização desta ferramenta pressupõe uma total confiança nas capacidades do Coq, pois se a correção e prova de teoremas do Compcert são realizadas no Coq, se este é incorreto então a correção do teorema não está assegurada.

2.2 O CASO DO COMPCERT C

O CompCert é um compilador otimizado e verificado formalmente que tem como alvo um grande sub-conjunto da linguagem C. Inicialmente desenvolvido e conduzido por Xavier Leroy em 2005 é especificado, programado e provado em Coq. Este suporta a maior parte dos tipos de dados do C (como *arrays*, estruturas e *unions*), com exceção de alguns tipos com maior precisão (*long double*) e *arrays* com tamanho variável.

Este compilador oferece garantias de correção sob o processo de compilação, garantias essas que ainda não foram quebradas, algo observado por John Regehr e o seu grupo de investigação que escreveram no *paper* Yang et al. [2011]:

> The striking thing about our CompCert results is that the middle end bugs we found in all other compilers are absent.(...)The apparent unbreakability of CompCert supports a strong argument that developing compiler optimizations within a proof framework

Nesta secção irá ser dado uma explicação, de acordo com Leroy [2009b], do processo de compilação incluindo os passos realizados pelo compilador desde o código fonte até à produção do código executável, dando um maior ênfase às fases que irão ser importantes para a realização do compilador certificado de circuitos.

O compilador CompCert é composto por três partes essenciais. A primeira parte consiste na conversão do código C em Árvores de Sintaxe Abstracta (AST) numa linguagem CompCert C. Aqui é feita a análise gramatical, bem como a análise dos tipos. Esta fase não está formalmente verificada, contudo, caso haja a ocorrência de um bug, este poderá ser facilmente detetáveis pois é possível fazer a transformação inversa: de AST CompCert C para um tipo sintático do C. No exemplo seguinte podemos ver as principais transformações que ocorrem de C para Compcert C.

```c
#include <stdio.h>
#define N 11
int main() {
  int i, sum = 0;
  for (i=0; i<N; i++) {
    sum += i;
  }
  return sum;
}
```

Listing 2.1: Código C

```c
(...)
int main(void){
  int i;
  int sum;
  sum = 0;
  for (i = 0; i < 11; i++) {
    sum += i;
  }
  return sum;
}
```

Listing 2.2: Código CompCert C

De notar que, para simplificar, no exemplo da linguagem intermédia CompCert C são omitidas a lista de chamadas a funções externas definidas antes do programa como *printf* e outras funções que lidem com tipos.

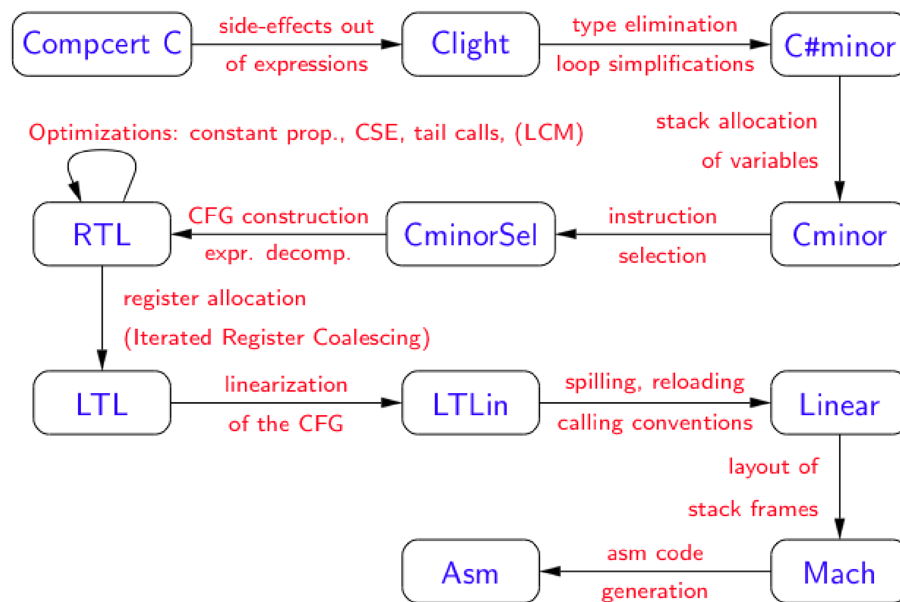


Figura 1: Estrutura global do CompCert C

A segunda parte representa a núcleo essencial do compilador, representa pela figura 1 e é aquela que se encontra provada corretamente pelo Coq. O compilador é composto em 8 linguagem intermédias e 14 passos. Todas estas linguagens têm uma semântica formal e a transformação entre cada linguagem está provada a sua preservação semântica. Aqui é feita a tradução desde o *Clight* para um *assembly* idealizado (sub-conjunto do *assembly* do PowerPC e IA32). O fato de existirem bastantes linguagem intermédias não é comum

para um compilador. Do ponto de vista de um programador, parece ser mais fácil ter o mínimo de linguagens e transformações possíveis. Contudo do ponto de vista da prova, é mais fácil definir um maior número de linguagem e transformações mais simples entre diferentes fases.

O *front-end* do compilador transforma as especificações das funcionalidades do código C em dois passos. Como é possível ver pela figura 1 esta fase envolve duas linguagens intermédias: *C#minor* e *Cminor*. Da fase do *Clight* para C#minor são eliminados os tipos e os ciclos são simplificados, que são substituídos por ciclos infinitos com *exits* nos respetivos blocos. Os tipos deixam de existir e para cada um são usados operadores aritméticos distintos. A fase seguinte, *Cminor*, é bastante similar à anterior só que retira da memória as variáveis a que não se acede ao respetivo endereço (com o operador &) - essas variáveis passam a ser consideradas variáveis locais definidas num ambiente próprio. Um exemplo das transformações referidas são os códigos seguintes que correspondem à tal transição.

```
int main(void)
{
  int i;
  int sum;
  sum = 0;
  i = 0;
  for (; 1; /*nothing*/, i = i + 1)
      {
    if (! (i < 11)) {
      break;
    }
    sum = sum + i;
  }
  return sum;
}
```

Listing 2.3: Código Clight

```
"main"() : int
{
  var 'i', 'sum';
  'sum' = 0;
  'i' = 0;
  {{ loop {
       {{ if ('i' < 11) {
            /*skip*/
          } else {
            exit 1;
          }
          'sum' = 'sum' + 'i';
       }}
       'i' = 'i' + 1;
     }
  }}
  return 'sum';
}
```

Listing 2.4: Código Cminor

A passagem entre *Clight* e *Cminor* envolve uma outra linguagem intermédia como ponto de transição, *C#minor*. Do *Clight* para esta são resolvidos todos os comportamentos dependentes de tipos e a conversão de declarações como ciclos e switch para representações mais simples, como é possível observar na transformação acima. De *C#minor* para *Cminor*, as funções têm exatamente uma variável endereçável e ocorre a passagem de variáveis para a *stack*.

O *back-end* do compilador começa por uma linguagem *CminorSel*, em que as suas características dependem do processador. Nesta fase são realizadas algumas otimizações na

combinação de instruções aritméticas e são criadas outras formas de operações, endereçamento e expressões condicionais. Esta parte torna-se importante no contexto do projeto pois é neste momento que se pretende a implementação de um oráculo(não certificado) capaz de prever, se possível, o nível de *unrolling* de um determinado ciclo. O passo seguinte transforma *CminorSel* para Register Transfer Language (RTL), construindo um grafo de controlo de fluxo. Nesta fase é possível a realização de várias otimizações como o *inlining* das funções, propagação de constantes, eliminação de código não necessário, entre outras.

O compilador contém ainda um conjunto de passos e transformações até ao final desta parte, que incluem a alocação de registos entre outras operações de baixo nível, contudo estas transformações não são relevantes para o projeto. A ultima parte do compilador, tal como a primeira não se encontra verificada. É realizada o *assembling* e *linking*, produzindo código executável.

2.3 CIRCUITOS BOOLEANOS

O produto final deste deste projeto é a geração de descrições de programas sob a forma de circuitos Booleanos, logo faz todo o sentido abordar este conceito.

É importante enunciar como estes poderão ser formalizados bem como entender a forma como os circuitos são importantes no contexto da criptografia, tal como o seu funcionamento geral. Nesta secção serão abordados estes temas e no final será referido algumas das formas como poderão ser classificados estes circuitos.

2.3.1 *Especificação de circuitos Booleanos*

Informalmente podemos definir um circuito Booleano como um modelo matemático que é definido pelas operações - ***gates*** - que contém e pelas dependências que existem entre as mesmas - **fios**. A definição apresentada por Vollmer [1999] afirmava que:

> *The logic circuit(...) is a directed acyclic graph whose vertices are labeled with the names of Boolean functions (logic gates) or variables (inputs). Each logic circuit computes a binary function* $f : B_n \rightarrow B_m$ *that is a mapping from the values of its n input variables to the values of its m outputs.*

Para além da enunciar a sua definição, Vollmer explicava que os circuitos executam **programas** ***straight-line***, ou seja, programas contendo apenas declarações de atribuição a variáveis. Assim este tipo de programas não contém ciclos ou ramificações. O exemplo seguinte ilustra como podem ser definidos este tipo de programas. O circuito exemplo trata-se de um *"full-adder"* apresentado na figura 2.

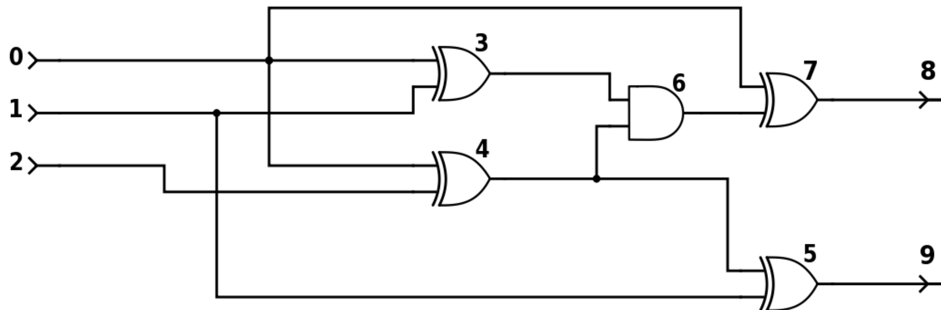


Figura 2: Esquema de um circuito - fulladder

Neste caso, este circuito representa a operação *fulladder*, esta é utilizada nas operações de adição de inteiros. O circuito recebe três inputs: os dois bits a somar e o *carry* de uma adição anterior, através de uma combinação de operações(and e xor) é operado o cálculo, retornado dois valores: um bit com o resultado e outro com o *carry* resultante da operação. Dado o circuito 2, um simples programa *straight-line* que traduziria esse circuito seria o seguinte:

$$
\begin{aligned}
x_3 &:= x_1 \oplus x_0 \\
x_4 &:= x_2 \oplus x_0 \\
x_5 &:= x_1 \oplus x_4 \\
x_6 &:= x_3 \wedge x_4 \\
x_7 &:= x_0 \oplus x_6
\end{aligned}
\tag{3}
$$

Nesta descrição de programa a cada input externo e a cada *gate* é atribuído um inteiro único. À enésima instrução é atribuída a variável x_n. Assim, se x_n é associado a um *gate* que combina os resultados produzidos no *gate* m e k com o operador $\odot$, então escrevemos uma operação de atribuição na forma de $x_n := x_m \odot x_k$.

Outra notação introduzida por Vollmer é apresentada de seguida, tendo como vantagens o fato de se aproximar mais de uma linguagem de programação de mais alto nível quando comparado com o conjunto de expressões anteriores.

```
0: INPUT x    // carryIn
1: INPUT y    // bit bA
2: INPUT z    // bit bB
3: XOR 1 0    // bA + carryIn
4: XOR 2 0    // bB + carryIn
5: XOR 1 4    // bA + (bB + carryIn)
```

```
6: AND 3 4    // (bA + carryIn) * (bB + carryIn)
7: XOR 0 6    // carryIn + ((bA + carryIn) * (bB + carryIn))
8: OUTPUT 7   // carryOut
9: OUTPUT 5   // bRes
```

Listing 2.5: Exemplo de uma possível descrição de um circuito booleano - fulladder

Assim podemos especificar que um programa *straight-line* é um conjunto de passos nos quais existe um passo do input, representado por *(s: INPUT x)*, um passo de output, representado por *(s: OUTPUT i)* ou um passo de computação *(s: OP i ... k)*. Aqui, s é o numero do passo e as palavras chaves INPUT, OUTPUT e OP identificam passos nos quais um input é lido, um output é produzido e uma operação OP é realizada, respetivamente. No enésimo passo de computação os argumentos de OP são os resultados produzidos nos passos $i, ..., k$. Logo, é necessário que estes passos precedam o enésimo passo, tal que $s \geqslant i, ..., k$

Como visto, cada passo deste tipo de programa computa uma função, que pode ser defina da seguinte forma. Seja g_s uma função computada pelo enésimo passo de um dado programa *straight-line*, f. Se o enésimo passo for um passo de input *(s: INPUT x)*, então podemos especificar que $g_s = x$. Se for um passo de computação *(s: OP i ... k)*, então a função é $g_s = OP(g_i, ..., g_k)$, onde $g_i, ..., g_k$ são funções computadas em passos anteriores. Se este o programa tem n input e m outputs, então podemos dizer que o programa traduz a função $f : B_n \to B_m$. Por fim, se $s_1, s_2, ..., s_m$ são os passos de output *(s: OUTPUT m)*, então dizemos que o programa é definido por $f = (g_{s_1}, g_{s_2}, ..., g_{s_m})$.

Esta definição de programa *straight-line* pode ser visto uma descrição de um circuito Booleano, que será o termo usado nesta dissertação para fazer referência a este tipo de programas.

2.3.2 *Circuitos de garbled*

Definido e formalizado o conceito de circuito Booleano é importante a compreensão de que estes são relevantes para determinados contextos e como foram adaptados par tal fim.

Os circuitos de *Garbled* foram apresentados em Yao [1986] como uma solução genérica para o problema de avaliação de uma função com segurança. Esta solução permite que dois participantes avaliem um circuito booleano com o seu input privado de forma a que a computação revele apenas o resultado da função.

O grande objetivo desta operação é a preservação dos dados de input de cada participante revelando apenas o output da operação sendo impossível para cada participante descobrir o input do outro. A função a ser realizada é representada por um circuito booleano onde os fios de input do circuito representam os inputs respetivos de cada participante.

Esta ideia foi introduzida por Yao como um protocolo de computação entre dois intervenientes, embora em trabalhos mais recentes se tenha estendido para um protocolo entre

vários participantes. Assim de forma a reunir uma ideia geral do modo de funcionamento geral vamos explicar o protocolo entre dois intervenientes.

No protocolo de Yao existem dois participantes (A e B) que pretendem computar a função $f(x_1, x_2) = (y_1, y_2)$ tal que A e B têm como input x_1 e x_2 e recebem como output y_1 e y_2, respetivamente. A função f é uma função Booleana que tem um representação sob a forma de circuitos onde x_1 e x_2 são os inputs dos circuitos. O participante A começa por cifrar o circuito c de forma a tornar o seu input privado. Para cada fio do circuito são escolhido dois valores(0 e 1). Depois para cada *gate* g, com inputs $b_1 \in \{0, 1\}$ e $b_2 \in \{0, 1\}$, os valores aleatórios correspondentes aos valores de input b_1 e b_2 são usados como chaves para cifrar o valor correspondente ao fio de output de g, ou seja, o resultado da computação de $g(b_1, b_2)$. Assim cada tabela de computação de um *gate* é também aleatoriamente permutado de forma a que não seja possível adivinhar as entradas da tabela pela sua ordem. Após a construção do circuito de garbled A envia-o para B juntamente com os inputs cifrados. Neste passo B avalia o circuito usando *1-out-of-2 Oblivious Transfer*[1] para cada um dos bits de input de forma a obter o valor cifrado correspondente de A. De referir o fato de B em nenhum caso tem acesso ao input privado de A. Durante o protocolo *Oblivious Transfer*, A envia as chaves usadas para cifrar o output de cada *gate*, de forma a que B apenas saiba as chaves, não os valores. Assim B usa esta técnica para avaliar todos os *gates*, terminando a avaliação do circuito obtendo o output da função que o envia para A.

2.3.3 *Otimizações de circuitos*

Desde a idealização dos circuitos de *Garbled* foram estudadas diversas formas otimizar as operações realizadas reduzindo o tempo de execução da avaliação do circuito.

Algumas dessas otimizações podem passar pela redução das tabelas de *Garbled*: *row-reduction* Beaver et al. [1990], evitar a avaliação de todas as linhas da tabela: *point and permute* Naor et al. [1999], entre outras. Contudo no contexto desta dissertação a otimização mais relevante foi apresentada por Kolesnikov and Schneider [2008]. Na sua investigação elaboraram uma nova construção de circuitos de *Garbled* para protocolos de Two-party secure function evaluation (SFE). Sucintamente, neste protocolo, *gates xor* são avaliadas *"for free"*, cujos resultados apresentam melhorias no desempenho nas implementações de circuitos de *Garbled*. Esta otimização *"free-xor"* implica um importante fato, em que a quantidade de dados transferido e o número de cifragens e de decifragens nos protocolos de circuitos de *Garbled* se foquem unicamente no número de *gates and* presentes no circuito Booleano e não nas *gates xor*. Assim, entre dois circuitos Booleanos representando a mesma função, aquele com menor número de *gates and* é preferível em detrimento do outro.

1 Protocolo onde B obtém de A a chave correspondente ao seu bit de input enquanto que A não sabe que chave ele enviou Even et al. [1985]

2.4 CBMC-GC

A necessidade da existência de uma ferramenta capaz de compilar programas para uma forma de circuitos booleanos, permitindo a sua utilização em protocolos Secure Two-Party Computation (STC), motivou a criação da ferramenta CBMC-GC por Holzer et al. [2012].

A ferramenta apresentada nesta secção apresenta os mesmos fins do compilador que se pretende desenvolver. Contudo esta não faz uso de uma compilação certificada, recorrendo a outros métodos para fazer a tradução para circuitos Booleanos. O CBMC-GC tem como principal objetivo transformar um dado código fonte C num circuito booleano otimizado. Esta tradução permite ao programador desenvolver funções em código C e, posteriormente, compilar programas para plataformas de STC.

2.4.1 *CBMC*

O CBMC-GC é baseado no *model checker* C-Bounded Model Checker (CBMC)[2] que permite a verificação de código ANSI C, em que o programa C é compilado juntamente com propriedades que se pretendem verificar produzindo fórmulas booleanas que são verificadas por um analisador de satisfatibilidade.

Para atingir o formato necessário para se realizar essa verificação o CBMC foi dividido em diversas fases. Inicialmente é realizado um conjunto de operações simplificação e transformação do programa incluindo a remoção de algumas diretivas e a normalização do programa num sub-conjunto da linguagem alvo. Após esta simplificação, existe uma tradução para um programa sem ciclos, ou seja, todos as declarações de controlo são substituídas por instruções *if-then-else* com saltos condicionais. De forma a perfazer um programa acíclico todos os ciclos são trocados por um sequencia de declarações condicionais aninhadas, e de maneira análoga, as chamadas recursivas também são expandidas n vezes. Assim para programas com um máximo n passos, este *unwinding* preserva a semântica do programa. Terminando o processo de remoção de ciclos, o CBMC transforma o programa para um formato Single Static Assigment (SSA), ou seja, num programa onde a cada variável é atribuída um único valor. Para tal, a cada atribuição de um variável é designado um novo índice permitindo assim uma representação de diferentes estados da computação para cada variável. Com este passo é possível a transformação do programa num conjunto de operações e equações sobre as variáveis. Num último passo, as variáveis são transformadas por vetores de bits, sendo que o tamanho das mesmas está dependente da arquitetura. As operações sobre as variáveis são traduzidas em funções Booleanas sobre as variáveis correspondentes. Por fim, estas funções (vistas como circuitos) são convertidas numa fórmula de conjunção de cláusulas Booleanas que representam a semântica do programa.

2 `http://www.cprover.org/cbmc/`

2.4.2 *Ferramenta CBMC-CG*

Uma vez revista os principais traços da arquitetura do CBMC é possível verificar a razão pela qual foi escolhido este software para servir de base ao CBMC-GC.

O CBMC-GC poderá ser divido em três passos até ao objetivo que é a geração de circuitos booleanos: **pré-processamento sintático**, **formação de circuitos** e **otimização de circuitos**. Numa primeira fase é feito e propagação de constantes para uma simplificação das computações. Através deste passo é possível a redução de número de *gates* do circuito final devido à simplificação de *array* quando é conhecido o endereço de uma certa posição em tempo de compilação, caso contrário, esta otimização não é possível e são utilizados circuitos multiplexados em leitura ou escrita em posições do *array*. Na segunda fase, são utilizadas e adaptadas as características do CBMC(utilização de circuitos para a representação de programas, como mencionado anteriormente) para a geração de circuitos que sirvam de input para protocolos de STC. Numa última fase ocorrem também algumas otimizações sobre o circuito resultante como a remoção de ramos de execução do programa que não estejam a ser utilizados, a instanciação de operações elementares como a soma ou a multiplicação que do ponto de vista da avaliação do circuito sejam mais rápidas (maior número de *gates xor* em detrimento de outras).

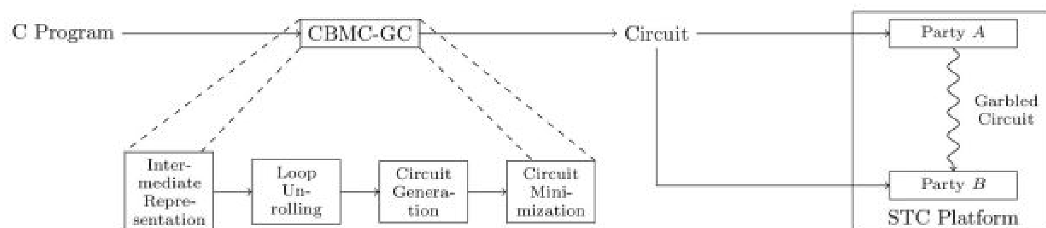


Figura 3: Arquitetura geral da ferramenta CBMC-GC

Como referido a utilização desta ferramenta pressupõe algumas limitações como o fato dos programas serem obrigatoriamente *bounded*, definindo um limite para o *unrolling* de ciclos, valor que pode ser passado explicitamente. Esta constante é utilizada no momento de realizar do *loop unwinding* e quando não for possível determinar o grau de *unwinding* necessário realizar para um ciclo na compilação do programa. Existem operações que ainda não são suportadas como operações de vírgula flutuante, bem como as operações aritméticas com apontadores ainda são bastante limitadas.

2 http://forsyte.at/software/cbmc-gc/

2.5 CONCLUSÕES

Neste capítulo foram abordados diversos assuntos que se tornam pertinentes para a compressão do contexto geral onde esta dissertação está inserida. Foram revistas abordagens e formas de alcançar uma compilação certificada de forma a compreender como é possível realizar uma compilador com tais características. Um exemplo destas práticas foi o CompCert C. Este será importante no desenvolvimento do compilador pretendido pois graças a este é possível o obter as garantias de compilação estudadas. Neste capítulo foi ainda revisto a forma pela qual são especificadas descrições de circuitos Booleanos, principal objetivo da investigação inerente a esta dissertação, bem como a sua importância e como estes são importantes em contextos da criptográficos. Por fim, foi estudado uma outra ferramenta, o CBMC-CG, que tem um objetivo muito semelhante ao compilador certificado que se pretende alcançar.

3

CIRCUIT DESCRIPTION GENERATION (CDG)

Neste capitulo será dado uma explicação sobre o circuito certificado de circuitos sobre o qual foi realizado o estudo e trabalho inerente a esta dissertação. Será dada um explicação sobre a arquitetura do Circuit Description Generation (CDG), seguindo-se uma explicação das várias fases de compilação dando relevo às mais importantes. É assim apresentado o compilador que foi realizado pelo grupo de investigação, na qual este projeto está integrado.

3.1 ARQUITETURA DO COMPILADOR

O Circuit Description Generation consiste num compilador certificado que converte um dado programa C em descrições de circuitos Booleanos. Esta ferramenta pode ser usada em contextos onde a computação é especificada em circuitos booleanos com a compatibilidade de output gerado ser compatível com a *framework FRESCO*[1]. Numa sub-secção mais avançada deste documento iremos dar uma explicação de como será o formato do output gerados e como estes se relacionam com esta *framework*.

A ferramenta é baseada na infraestrutura do compilador certificado CompCert, sendo dividido em dois principais componentes:

- CDG Frontend - cuja tarefa é converter o programa fonte numa representação de uma linguagem intermédia. Sendo que os programas nesta linguagem estão processados e restritos para que seja possível atribuir-lhes uma representação de um circuito Booleano;
- CDG Backend - engloba a formalização da especificação do circuito Booleano pretendido e as transformações e estados até ao circuito final pretendido.

O esquema 4 representa os traços e componentes gerais do compilador certificado de circuitos desde os vários passos de compilação bem como novas linguagens intermédias implementadas para a geração do circuito. A figura representa o percurso pelo qual um

1 *Framework Java* para computação segura e eficiente desenvolvida no âmbito do PRACTICE

dado programa C terá de seguir para que seja transformado numa descrição de um circuito Booleano.

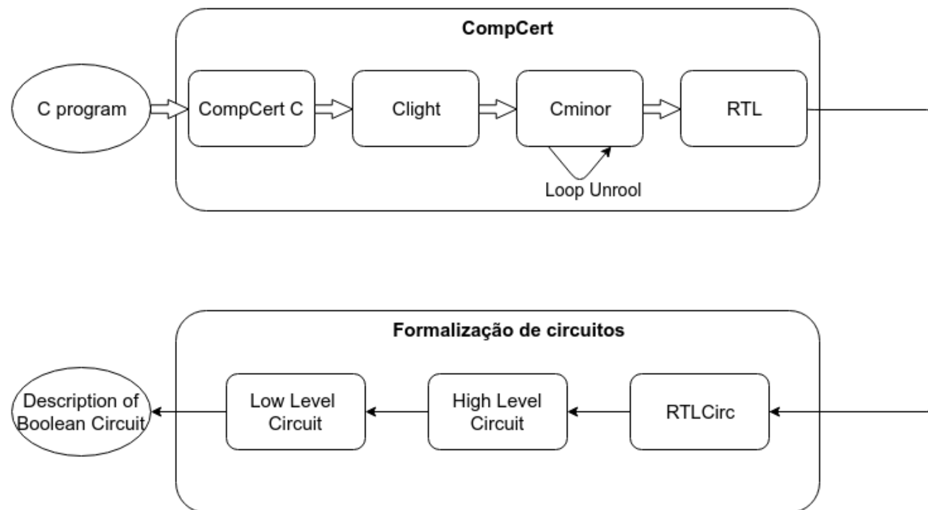


Figura 4: Arquitetura do compilador certificado de circuitos

Tal como já abordado em capítulos anteriores a ferramenta de geração de circuitos certificados foi construida com base no CompCert. As características certificadas deste compilador certificado são aproveitadas para a construção de parte do compilador, confiando nos passos de compilação até à linguagem intermédia RTL. Ainda na fase de compilação do CompCert, mais concretamente no Cminor, é adicionado um oráculo de previsão de grau de *unrolling* de ciclos. Este oráculo será útil em fase mais avançadas da compilação pois tornará a prova da preservação da semântica do programa mais fácil.

Uma vez ao nível do RTL, os ramos condicionais são traduzidos para uma estrutura de controlo de fluxo, sendo que posteriormente é gerado uma abstração de circuitos de mais alto nível, em que estes servem de apoio à geração de circuitos num formato mais elementar.

3.1.1 *Restrições do compilador*

De forma a tornar possível a geração de tais circuitos é necessário impor duas restrições sob os programas C fonte. Primeiro, o comportamento do programa tem de ser completamente determinístico, dependendo apenas dos inputs do programa. Outra restrição a ser feita é que o programa deve terminar para todos os possíveis dados de input e, para isto ser atingido o programa deve permitir que todas as funções possam ser estaticamente *inlined*, os ciclos possam ser estaticamente e completamente *unrolled* e a não existência de alocação de memória dinâmica.

3.2 FRONT-END

Tal como referido anteriormente devido às suas características certificadas foi usado o CompCert C como base para o compilador certificado de circuitos. Dadas as suas garantias de compilação certificadas, este assume-se como um compilador confiável para a transformação do programa até a um estado conveniente para a geração dos circuitos pretendidos.

O Compcert envolve todo um processo de compilação desde o código C até ao código executável final. Como é óbvio apenas foram aproveitadas e utilizadas as fases deste compilador certificado até ao momento(RTL) em que é útil e conveniente para a transformação em representações de circuitos Booleanos.

É nesta fase que o programa apresenta determinadas características como instruções de controlo de fluxo muito próximas da representação de um código *assembly*. Portanto faz sentido que nesta fase da compilação seja iniciada a transformação para descrições de circuitos Booleanos.

Ainda nesta fase de compilação foi adicionado um oráculo de previsão de grau de *unrolling* de um determinado ciclo de forma a otimizar etapas seguintes do processo de compilação.

3.2.1 *Validação Estrutural*

Antes da passagem para a formação de circuitos Booleanos foram adicionadas algumas verificações sobre o RTL resultante, modificando o *type-checker* para restringir o programa inicial. Assim para a geração de circuitos apenas aceites circuitos que apresentem determinadas restrições. Entre essas imposições estão as seguintes:

- Existência de uma função - *main*;
- Programa iniciado por uma declaração de uma sequência de inputs;
- Programa finalizado por uma declaração de uma sequência de outputs, com um único ponto de saída;
- O corpo do código está restrito a:
  - Exclusão de operações com chamadas externas;
  - Apontadores a zonas definidas na memória.

3.3 BACK-END

Nesta secção será explicada como foi traduzida a passagem de código sob a forma da linguagem intermédia RTL até à formação de descrições de circuitos Booleanos. Para tal transformação foram definidas duas linguagens intermédias, Register Transfer Language Circuits (RTLC) e High Level Circuit (HLC). Ambas as linguagens bem como as transições entre elas foram desenvolvidas e provadas em Coq. A geração de descrições de circuitos foi realizada em OCaml, não estando portanto esta passagem verificada formalmente, contudo permite que o resultado da transformação final sejam descrições compatíveis com determinados protocolos criptográficos.

A especificação e transformação das linguagens intermédias (RTLC e HLC) enunciadas neste capitulo foram realizadas no âmbito do projeto PRACTICE e será explicado sucintamente a forma como foram desenvolvidas pela equipa de investigação. O mesmo acontecerá para a parte final do compilador.

3.3.1 *Register Transfer Language Circuits (RTLC)*

Nesta sub-secção será apresentada um linguagem intermédia adicional que foi adicionada após a verificação estrutural do programa resultante da compilação do CompCert. Esta nova linguagem é uma adaptação do RTL gerado pelo CompCert.

Assim apresentamos a linguagem intermédia Register Transfer Language Circuits (RTLC) que é uma variante da linguagem RTL onde a dinâmica de controlo de fluxo é substituída por guardas em cada instrução. Estas guardas representam o conjunto de condições necessárias para que uma dada instrução seja atingível.

As guardas referidas funcionam como *path-conditions* de uma instrução. Como a construção desta informação é realizado à custa das *path-conditions* das instruções antecessoras da mesma, é necessário que previamente tenham sido recolhido os antecessores de todas as instruções. A representação e definição dos principais tipos do RTLC é apresentado de seguida.

```
Definition code := list (pcond*instruction).

Inductive instruction: Type :=
  | Itest: node → condition → list reg → instruction
  | Iphi: node → list (pcond*reg) → instruction
  | Iop: operation → list reg → reg → instruction
  | Iload: memory_chunk → addressing → list reg → reg → instruction
  | Istore: memory_chunk → addressing → list reg → reg → instruction
```

Listing 3.1: Definição do tipos das operações em RTLC

Nesta linguagem são eliminadas as instruções condicionais (*Icond*) sendo substituídas por instruções de teste (*Itest*) que contêm uma determinada condição a verificar. Como dito anteriormente a cada instrução é associado uma *path-condition* - *pcond*.

Para a representação de circuitos Booleanos, uma propriedade intermédia a realizar é a **passagem do código para SSA**. Neste formato é garantido que a cada variável é realizada apenas uma atribuição. Com a informação em cada instrução das *path-conditions* é possível e simplificada a passagem para SSA.

Para a resolução de múltiplas atribuições a uma variável é adicionada a instrução *Iphi*. Assim, quando é realizada uma leitura numa variável é verificado se para esta é necessário criar um novo estado da variável. A criação ou não deste novo estado depende das escritas anteriores à variável e das *path-conditions* onde essa escrita foi realizada. Portanto, quando essa leitura na variável não consegue determinar um e um só estado da variável é adicionado ao código a instrução *Iphi*, criando uma nova definição da variável.

Formato Single Static Assigment (SSA). *Com a realização das operações mencionadas anteriormente é atingido um estado do programa em que a cada variável é atribuído um e um só valor, correspondente ao valor dado pelo phi-node. A introdução de instruções do tipo de phi-nodes permitiu a resolução de múltiplas atribuições numa variável, estando o controlo de fluxo das diversas variáveis tratado de forma a ser possível a transformação do programa num circuito.*

Atingido o programa este formato, está facilitado a passagem para circuito Booleanos, uma vez que as variáveis utilizadas poderão ser representadas por fios do circuito que representa o programa.

3.3.2 *High Level Circuit (HLC)*

Uma vez com o programa sob a forma de SSA é necessário a conversão para um programa que esteja mais próximo conceptualmente de um circuito Booleano. Assim é apresentada a linguagem intermédia HLC, que apresenta características muito próximas de um circuito Booleano, com as suas operações exprimidas na forma de *gates* e estando também definidas ligações entre as diversas operações. Esta linguagem encontra-se definida em Coq, sendo que a transformação realizada para alcançar esta linguagem foi igualmente provada em Coq.

Para se atingir um estado, apresentado de seguida, onde as operações estão definidas por fios e *gates* é necessário que haja uma substituição não só das variáveis mas também das localizações na memória utilizadas, por conjunto de fios que traduzam o mesmo tipo

de informação. Esta **substituição destas variáveis por fios** é importante pois é desta forma que será expresso o fluxo de dados dentro de um circuito Booleano.

As definições seguintes representam a forma como está definido o programa.

```
(** Wire := (entryNum * pos) *)
Definition Wire := (N*N)%type.
Definition Conn := seq Wire.

Record Gate := { gate_in_arity : N
               ; gate_out_arity : N
               ; gate_name : string
               ; gate_args : seq int
               ; gate_en : bool
               ; gate_eval : seq bool → seq bool
               }.

Record GEntry := { gate: Gate
                 ; conn: Conn
                 }.

Record Circuit := { inputs: seq N
                  ; outputs: Conn
                  ; gates: seq GEntry
                  }.
```

Neste ponto é definido um tipo abstrato de **Gate** que servirá de apoio para guardar os diferentes tipos de operações. Para expressar as ligações entre *gates* ou *outputs* é utilizado um tipo **Conn**, que é uma sequência de **Wire**, que representa um tuplo com o número identificador da *gate* e a posição no conector. A definição de um circuito contém uma sequência de inteiros - inputs, um conjunto de fios - outputs e uma sequência de entradas compostas pela informação das *gates* e o conjunto de fios associados.

Uma propriedade adicional no formato deste circuito é que a *gate* na posição zero está reservada para a representação de dois fios especiais: os valores de verdade - (0,0) é *False* e (0,1) é *True*. Esta informação é útil para registar se uma determinada operação será realizada ou não, podendo reduzindo o tamanho geral do circuito.

É nesta fase que se realiza a concretização das *gates* sendo especificado para todo o tipo de instrução as características apresentadas anteriormente como o número de inputs, outputs, os parâmetros que a operação possa ter, a semântica da operação, o seu nome,...

Com esta representação de informação é possível definir um circuito Booleano composto por um conjunto de operações interligadas e bem definidas que poderão ser vistas como *gates* complexas que representam uma determinada função lógica.

3.3.3 *Low Level Circuits*

De forma a terminar a transformação de um programa C para uma descrição de um circuito Booleano que possa ser utilizado em determinados contextos, é necessário converter o formato desenvolvido e provado em Coq para outro com um determinado formato mais específico.

Algum do trabalho realizado no âmbito deste projeto inseriu-se nesta secção, tendo sido realizada uma ferramenta capaz de gerar as tais descrições de baixo nível que poderão ser utilizadas em determinados protocolos de computação segura. Apesar de ter sido desenvolvida a tal ferramenta, atualmente no compilador encontra-se implementado um outra forma de geração das descrições de circuitos Booleanos, que irá ser abordada de seguida.

Esta última fase do compilador de circuitos não foi realizada em Coq, mas em OCaml. Foi tirado o proveito das capacidades do ferramenta Coq, para extrair os tipos da linguagem abordada na secção anterior. É portanto necessário realizar a **conversão para OCaml** e uma vez que os tipos entre ambas as linguagens são bastante idênticos, para uma tradução em OCaml foram apenas substituídos alguns tipos correspondentes de uma linguagem para a outra, por exemplo de *N* para *int*, de *seq* para *list*, etc.

Uma vez com a informação toda traduzida em OCaml, é realizada a **instanciação das *gates***. Assim, são declaradas as *gates* mais elementares: *xor* e *and*. Através destas, são construídas e definidas as operações que se pretendem representar, sendo que a composição destes circuitos é utilizada para a expressão de operações cada vez mais complexas. Esta composição envolve a interligação de fios entre as várias *gates* definidas.

No exemplo 3.2 encontra-se definido um circuito que simula o comportamento de uma *gate* **not** e é instanciada à custa de outra *gate* **xor**. Na definição da *gate xor* está definida a semântica da mesma, sendo que, juntamente com a *gate and*, são as únicas que precisam que a sua semântica esteja definida pois o todas operação são compostas pelo conjunto das mesmas.

```
let g_XOR = { hlgname = "XOR"
       ; hlgargs = []
       ; hlgsem = Some (fun l ->
              match l with
              | [x;y] -> [if x then not y else y]
              | _ -> internal_error "gXOR_sem: wrong arguments!")
       ; hlgin = 2
       ; hlgout = 1 }
let gc_NOT = { hlcin = [1]
        ; hlcgates = [(g_XOR, [(1,0);(0,1)])]
        ; hlcout = [(2,0)] }
```

Listing 3.2: Exemplo da instanciação de um circuito que representa a função NOT

Registadas e implementadas as *gates* necessárias para a produção dos circuitos pretendidos é realizada a **expansão das *gates***. Sendo que processo é automático existindo uma correspondência dos nomes das *gates* de acordo com o nome da instrução definido nos passos do CompCert. Assim, de forma a manter a coerência das operações sob as *gates* do circuito alvo, são realizadas atualizações do identificadores dos fios de input de cada *gate* tendo em conta o contexto geral do circuito. O formato final produzido trata-se de uma descrição idêntica aquela apresentada na secção 2.3.1 onde foi especificado a formalização de circuitos Booleanos.

Atualmente no compilador estão a ser suportadas operações de qualquer tipo de comparações, bem como quase todas as operações unárias presentes no CompCert. Em relação às operações binárias apenas foram definidas algumas como subtrações, adições e *and/xor/or* de 32 bits.

Durante a geração do circuito neste formato, de forma a tornar a avaliação posterior do circuito mais eficiente, são realizadas algumas operações que poderão permitir a redução do tamanho geral do circuito. Esta **simplificação do circuito** consiste na remoção de *gates* que sejam inacessíveis, ou seja, independentemente da avaliação realizada essa *gate* nunca será avaliada. Outras otimizações como a reutilização de fios que representem o output de uma mesma operação com o mesmo input.

É importante ressalvar que todas as operações feitas foram cobertas por alguns testes que verificam a coerência e correção dos resultados produzidos. Assim estão presentes alguns testes aleatórios sob operações mais simples, e posteriormente para determinadas funções mais complexas é possível fornecer de alguns testes como o input e o output desejado.

Nesta ultima fase é possível ainda realizar uma transformação para um circuito num formato convencionado pela Universidade de Bristol. A descrição deste formato, designado por **formato de Bristol**, encontra-se descrita em Tillich and Smart [2015]. De acordo com este formato textual, um circuito é descrito por um ficheiro em que:

- Uma linha define o número de *gates* e de seguida o número de fios num circuito;
- Noutra linha, dois números n_1 e n_2 de fios correspondentes aos inputs da função descrita pelo circuito - a função contém no máximo dois inputs, e em caso de não houver um segundo argumento o segundo número de inputs permanece a zero;
- Na mesma linha é ainda representado número de fios no output n_3;
- Os fios estão ordenados tal que os n_1 fios correspondem aos fios de input do primeiro argumento, os seguintes n_2 fios correspondem ao segundo valor de input. Os últimos n_3 fios correspondem aos outputs do circuito;
- De seguida são listados as *gates* com a seguinte informação: número de fios de input, número de fios de output, lista de fios de input, número de fios de output e nome da *gate (xor, and ou inv)*.

Tanto no formato de Bristol como no produzido anteriormente são produzidos os elementos que permitem uma classificação do circuito em termos de complexidade de avaliação, visto que este é apenas constituído por *gates* do tipo mais elementar - *and, xor, inv*.

4

CONTRIBUIÇÕES

Neste capitulo vai ser detalhado todo o trabalho prático realizado em torno da investigação desta dissertação. Para a elaboração do CDG, a contribuição realizada consiste na construção de dois componentes. Um é a realização de um oráculo capaz de prever a nível de *unrolling* de um determinado ciclo e a outra contribuição é a realização de uma ferramenta para auxiliar na geração de descrições de circuitos Booleanos a partir de outras descrições que se encontram num grau de abstração maior: circuitos compostos por *gates* complexas.

4.1 ORÁCULO DE PREVISÃO DE NÍVEL DE *unrolling* DE CICLOS

A funcionalidade apresentada nesta secção é capaz de prever, dado um determinado ciclo, o grau de desdobramento desse ciclo, ou seja, o número de iterações que esse ciclo executa. Este oráculo funciona em tempo de compilação portanto tem de ser capaz de distinguir se é possível ou não a realização desta previsão.

No processo de compilação de um programa o desdobramento de ciclos é necessário para a construção de um grafo de fluxo que traduza o comportamento do programa. O passo de **Loop Unroll** adicionado ao CompCert está responsável por essa eliminação de ciclos. Atualmente, no contexto do projeto PRACTICE este passo está realizados em dois passos:

- Realização de *unrolling* do ciclo num número arbitrário de iterações, permanecendo o ciclo desenrolado;
- Transformação que permite ficar com apenas a parte do ciclo que após o desenrolar do ciclo é atingível.

O oráculo desenvolvido foi integrado na primeira transformação enumerada. No protótipo do compilador fornecido o número de *unfolds* para cada ciclo é apenas uma função constante que apenas devolve um determinado argumento passado na linha de comandos.

A necessidade desta funcionalidade prende-se com o fato de a passagem deste argumento como valor por defeito origina a criação de um número elevado de iterações que poderão não ser utilizadas e que posteriormente terão de ser removidas. Sendo assim, de forma a

otimizar este processo foi implementado um oráculo capaz de tentar prever com exatidão o número de *unfolds* necessários para desenrolar um determinado ciclo, originando uma representação de código mais reduzida.

4.1.1 *Formato da linguagem ambiente*

Num dos capítulos anteriores foram explicados os vários passos de compilação do CompCert C tendo como objetivo o de evidenciar as principais características de cada linguagem intermédia. Mais em concreto, foi explicado a linguagem intermédia *Cminor*. A realização do oráculo desenvolvido encontra-se numa linguagem intermédia bastante semelhante a essa, o **CminorSel**. De seguida vão ser explicadas as principais características desta linguagem dada a sua relevância para a realização do oráculo.

A passagem de *Cminor* para *CminorSel* é conhecida por *Instruction Selection*. Esta linguagem é a primeira que está dependente da máquina reconhecendo sequências de operações que a máquina alvo pode transformar numa instrução. Este passo também converte os *loads* e *stores* de acordo com o modo de endereçamento da arquitetura alvo, portanto a semântica do *CminorSel* é personalizada para uma determinada arquitetura.

Para a realização do oráculo é necessário conhecer a forma como são representados os ciclos bem como está organizada todo o programa ao nível do *CminorSel*. A declaração seguinte representa a forma pela qual são representadas as várias operações nesta linguagem intermédia:

```
Inductive stmt : Type :=
  | Sskip: stmt
  | Sassign: ident → expr → stmt
  | Sstore: memory_chunk → addressing → exprlist → expr → stmt
  | Sseq: stmt → stmt → stmt
  | Sifthenelse: condexpr → stmt → stmt → stmt
  | Sloop: stmt → stmt
  | Sblock: stmt → stmt
  | Sexit: nat → stmt
  | Sswitch: exitexpr → stmt
  | Sreturn: option expr → stmt
  | Slabel: label → stmt → stmt
  | Sgoto: label → stmt
 ...
```

As declarações presentes no *CminorSel* incluem atribuições a variáveis locais (*Sassign*), armazenamento em memoria (*Sstore*), operações condicionais (*Sifthenelse*), ciclos infinitos (*Sloop*), saída de blocos de instruções (*Sexit*). Assim é possível observar a forma como são

construidos as várias formas de representação de ciclos em código C (tanto ciclos *for* ou *while* e as diferentes variantes de declarações de variáveis nesses dois tipos).

A transformação de 4.1 para 4.2 representa uma das formas como são especificados os ciclos na linguagem intermédia *CMinorSel*.

```
for(i = 0; i < 11; i++) {
    ...
}
```

Listing 4.1: Exemplo de um ciclo em C

```
'i' = 0;
{{ loop {
    {{ if ('i' < 11) {
        } else {
            exit 1;
        } ...
    }}
        'i' = 'i' + 1;
    }
}}
```

Listing 4.2: Exemplo da representação textual em CMinorSel

Como podemos observar o ciclos em *CMinorSel*, a atribuição inicial da variável de controlo é passada para o conjunto de instruções fora e anteriores ao ciclo. A condição do ciclo é representada na primeira operação condicional depois da declaração do ciclo e valor de incremento é representado como uma atribuição no final do ciclo. Esta posição da operação de incremento poderá diferir se o ciclo for um *for* ou *while*, contudo a operação da atribuição encontra-se sempre dentro da declaração do ciclo.

4.1.2 *Processo de realização do oráculo*

Visto as principais caraterísticas do formato desta linguagem intermédia é possível explicar como foi elaborado o oráculo em OCaml. Tal como dito anteriormente este oráculo vêm substituir uma funcionalidade que fornece um argumento em vez de uma tentativa de previsão do grau de *unrolling*. Tendo esta tradução em vista o oráculo tem apenas dois resultados possíveis: uma correta previsão do número de iterações de um determinado ciclo ou a incapacidade de determinar esse tal número, sendo o resultado o número que foi passado como argumento, valor *default*.

De seguida serão apresentados as etapas seguidas até ao cálculo final do número de iterações em cada ciclo:

1. Identificação do ciclo;
2. Acumulação e registo da sequência de instruções até ao dado ciclo;
3. Extração da operação condicional;

   a) Identificação da variável de controlo e respetivo limite;
   b) Procura nas instruções anteriores acumuladas da última atribuição a essa variável.

4. Identificação de uma atribuição à variável de controlo dentro do ciclo;
5. Verificação da existência e coerência dos valores;
6. Cálculo do número de iterações tendo em conta os valores extraídos.

Explicando os passos realizados, o processo é iniciado com a identificação do ciclo pretendido, sendo que enquanto se percorre as instruções até ao mesmo estas são guardadas para utilização futura. Descoberto o ciclo é necessário identificar a variável de controlo, que como vimos anteriormente, encontra-se na primeira operação condicional após a declaração de um ciclo. Dentro da operação condicional apenas são permitidas operações simples, ou seja, que tenham apenas uma variável(se isto não acontecer não é atribuído nenhum valor ao campo da variável condicional). Com esta informação, a variável de controlo e respetivo valor condicional, é procurado no histórico de instruções acumuladas anteriormente a última atribuição a essa variável, mais uma vez apenas são suportadas atribuições simples. Por fim, é procurado por uma atribuição dentro do ciclo a essa variável, que corresponde ao valor de incremento pretendido. Se ocorrerem mais do que uma atribuição a essa variável dentro do ciclo, o oráculo rejeita a previsão de valor.

Para fazer o cálculo final do número de iterações de cada ciclo são necessários três valores: o valor inicial, da condição e o incremento. Para o cálculo final é necessário que estes contenham algum valor (uso do tipo *option*) e que os valores entre si tenham lógica. Por exemplo, se a diferença entre o valor da condição e o inicial for maior que zero o incremento também tem de ser maior que zero. Assumindo que estas condições são atingidas, é realizado o cálculo seguinte:

$$f(init, cond, inc) = \lceil \frac{|cond - init|}{|inc|} \rceil$$

Tal como já abordado ligeiramente, o oráculo desenvolvido foi desenvolvido para apenas prever o número de iteração quando há certezas de que tal valor é correto. Logo quando há operações que possam alterar o comportamento de uma variável, o oráculo devolve o valor *default* passado como parâmetro.

4.2 FERRAMENTA GERADORA DE CIRCUITOS

Nesta secção será apresentada a ferramenta desenvolvida para servir de apoio à geração de circuitos Booleanos resultantes da compilação certificada do compilador apresentado. Esta ferramenta é uma alternativa à abordagem atual presente no compilador apresentada na secção 3.3.3. O objetivo desta ferramenta é transformar descrições de alto nível de circuitos Booleanos, abordadas na secção 3.3.2, noutras onde os circuitos sejam expressos num formato mais elementar possível.

A figura 5 representa a arquitetura geral da ferramenta desenvolvida. De forma concisa, a ferramenta recebe um ficheiro que contém uma descrição de um circuito num formato de alto nível e, através de outros circuitos, construidos à mão e que representam determinadas operações a realizar, é construido uma descrição de um circuito final em que o seu formato é o mais elementar possível.

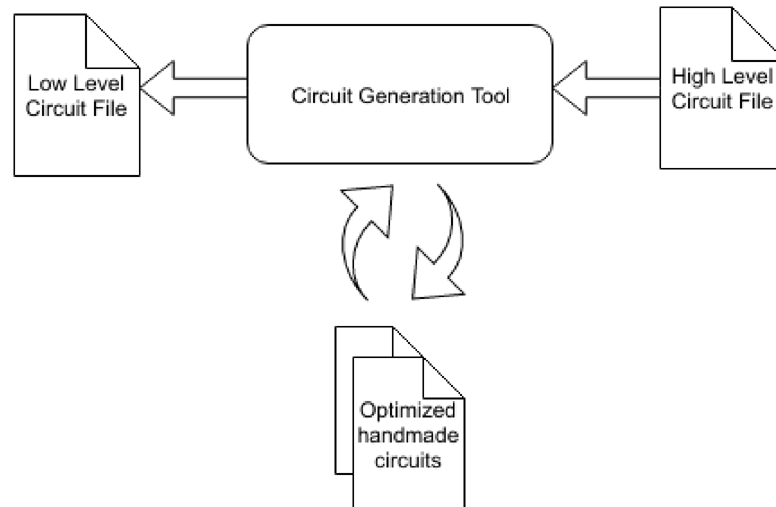


Figura 5: Arquitetura da ferramenta geradora de circuitos de baixo nível

O circuito fonte é composto por uma lista de operações que agregadas compõem a funcionalidade do circuito final, como tal, é necessário exprimi-las sob a forma de *gates* lógicas (*and*, *xor*). Juntamente a cada uma destas operações estão os fios de input das operações, sendo que numa secção futura será apresentado o formato destes circuitos. Estas operações podem exprimir outros circuitos Booleanos e no limite serão apenas constituídas pelas tais *gates* elementares enunciadas anteriormente.

Os circuitos realizados de apoio à ferramenta traduzem determinadas operações como adição, multiplicação, seleções e atualizações em *array*, etc. Cada um destes circuitos é descrito por outros sub-circuitos, sendo que estes também poderão ser também representados por outros circuitos até que seja possível a representação do circuito principal apenas com *gates* elementares.

4.2.1 *Low Level Boolean Circuit*

O principal propósito da construção do compilador certificado de circuitos é a geração de descrições capazes de serem utilizadas em protocolos de computação segura. Estas descrições tem de constar num formato simples e elementar que sejam compatíveis e equivalentes com outros formatos usados em determinados protocolos. Assim, vamos apresentar um dos formatos de saída possíveis da ferramenta e consequentemente do compilador certificado.

A especificação do circuito pretendido pode ser descrito tendo em conta o seguinte formato:

```
k: INPUT
n: GATE <id> <id>
m: OUTPUT <id>
```

O circuito final é constituído essencialmente por três blocos de informação diferentes - uma região de inputs, outra de *gates* e uma final com os outputs. Cada linha desta descrição é inicializada por um inteiro que funciona como identificador da linha.

A descrição deste circuito é iniciada por um conjunto de k inputs que correspondem precisamente aos fios de input do programa. Tal como mencionado anteriormente estes fios são identificados por um inteiro. Depois da declaração de todos os fios de input, é declarado o conjunto de *gates* que realizam a lógica do programa. A descrição de cada *gate* é acompanhada por um ou mais identificadores que correspondem aos fios de input da *gate* em questão. Estas *gates* poderão ser de diversos tipos podendo aceitar um ou dois fios de input. Por fim, no formato da descrição deste circuito Booleano, são declarados as declarações de output. Cada fio de output é acompanhado por um identificador que corresponde ao fio de output de uma *gate* ou em determinados casos a um fio de input.

Esta descrição é compatível com outras implementações que servem de apoio a testes sob implementações de protocolos MPC e FHE. Caso disto são os circuitos implementados por Tillich and Smart [2015], que a descrição utilizada por eles é equivalente à descrita nesta secção, tal como explicado na secção 3.3.3.

4.2.2 *High Level Circuit file description*

Nesta sub-secção é apresentado o formato que é tratado pela ferramenta. Esta descrição serve de suporte não só ao ficheiro de entrada mas também aos ficheiros que contêm as operações a realizar. Os circuitos que contém as diversa funcionalidades a realizar foram elaborados à mão, sendo cada um definido à custa de outros circuitos mais simples.

Este formato caracteriza-se pela sua simplicidade mas ao mesmo tempo tem a capacidade de exprimir operações sob a forma de circuitos Booleanos de forma prática e intuitiva. Assim, de seguida será apresentada o formato destas descrições:

```
k: INPUT <bus-width>                                        | INPUT SECTION
n: <gate> [ <bus-spec1> ; ... ; <bus-speck> ]               | GATE SECTION
m: OUTPUT [ <bus-spec1> ; ... ; <bus-speck> ]               | OUPUT SECTION
```

Estes circuitos são descritos como *arrays* de *buses* interligados com *gates* complexas. Os *buses* referidos são barramentos constituídos com um conjunto de fios. Cada *gate* poderá ser construida a partir de outros circuitos mais simples, e estes circuitos contêm *gates* mais simples que são construidos também a partir de outros circuitos até ao ponto em que todos os circuitos alcancem todas as *gates* no estado mais elementar (*and, xor*).

A secção de inputs é representada pelas primeiras posições do *array* e cada posição contém a informação do tamanho de input - número de fios de input. Cada *gate* tem uma aridade fixa no que diz respeito ao número de fios de input e output. Assim, na especificação de um *gate*:

```
n: <gate> [ <input-wires> ]
```

a concatenação de todos os fios de input deve corresponder à aridade de entrada da *gate*. Do mesmo modo, *n* funciona como um identificador de um *bus* como uma determinada aridade de saída.

Um *bus* simples é identificado como um tuplo (<bus-id>, <start-bit>, <length>), onde <bus-id>é esperado que seja um *bus* que ocorra antes do *gate* atual e com pelo menos *start_bit+length* bits.

De maneira análoga ao que acontece na especificação de *gates*, a secção de output é composta por um conjunto de posições que contém um conjunto de *buses* que correspondem aos fios de output do circuito em questão.

Este tipo de descrição de circuitos é útil pois assim é possível a construção de circuitos Booleanos de forma simples e sistemática. Tanto o circuito de entrada da ferramenta como os ficheiros auxiliares têm de se encontrar de acordo com esta descrição. Relativamente aos ficheiros auxiliares que correspondem aos ficheiros que contêm as descrições de determinadas operações foram realizados de forma a que estes sejam o mais eficientes possíveis. Esta medida de eficiência, já abordada anteriormente, prende-se no fato de realizar as operações desejadas com o menor número de *gates and* possíveis.

Estes circuitos traduzem representações de determinadas operações que poderão ser otimizadas com vista a uma melhor performance do circuito numa avaliação geral do circuito.

4.2.3 *Funcionamento da ferramenta*

Revistos os vários tipos de descrições de circuitos Booleanos que envolvem a ferramenta desenvolvida, vamos abordar a metodologia utilizada para a transformação entre descrições. A ferramenta desenvolvida foi realizada na linguagem OCaml. Nesta sub-secção vamos abordar as diversas etapas que foram tomadas para chegar ao produto final levando em conta os aspetos mais relevantes da implementação da ferramenta.

Leitura e parsing do circuito

As diversas descrições de circuitos Booleanos que são o alvo das transformações a realizar terão de ser encontrar descritas num formato textual. É portanto necessário que o programa permita a leitura destes ficheiros de forma sistemática e correta, logo para facilitar a leitura e *parsing* do mesmo foi criado um *parser* capaz de converter as descrições de HLC para estruturas de dados de forma a aplicar as transformações desejadas. De maneira a realizar esta leitura foram utilizados geradores de *parsers* semelhantes a **lex** e **yacc** em OCaml: o **ocamllex** e **menhir**[1], respetivamente. Realizada uma gramática que obedeça ao formato de circuitos de alto nível já abordados, é também necessário um *lexer* para a conversão do texto de input numa *stream* de *tokens*.

Como tanto o ficheiro que contém a descrição do circuito principal como as outras descrições que servirão para realizar as expansões necessárias contém o mesmo formato, logo a metodologia de como são lidos é exatamente idêntica.

Tratamento de informação

Desenvolvidas e captadas as produções e regras do formato apresentadas na secção 4.2.2 pela gramática desenvolvida, o próximo passo importante realizado foi alcançar uma forma de armazenar a informação de maneira a ser possível uma transformação para um formato mais elementar. Assim apresentamos a estrutura de dados 4.3 que representa a forma como foi captada a informação das descrições de circuitos.

```
type circuit = {
  mutable name:          string;
  mutable inputs:        (int * input)  list;
  mutable gates:         (int * gate)   list;
  mutable outputs:       (int * output) list;
  mutable wire_input:    bus list;
  mutable new_gate_pos:  (int, int) Hashtbl.t;
```

1 http://gallium.inria.fr/~fpottier/menhir/

```
}
  and input = int
  and output = bus list
  and gate = | Elem of (string * bus list)
             | Circ of circuit
```

Listing 4.3: Tipo de dados que armazena as descrições de circuitos Booleanos

Esta estrutura de informação guarda os três tipos de entradas possíveis: inputs, *gates* e outputs, sob a forma de várias lista de associação em que o primeiro valor é o identificador da linha respetiva. No caso de ser um input este é apenas associado a um inteiro que indica o número de fios de input. Se for um output, o identificador é associado a uma lista de *buses* que correspondem aos fios de output do circuito. Se for um *gate* poderá ter duas construções: Caso seja um *gate* elementar será guardado o seu nome bem como os fios de input desta *gate*. Caso seja um circuito que necessite de ser expandido é inicializada uma nova estrutura *circuit* onde é atribuída à variável *wire_input* os *buses* que servem de input a este novo circuito.

Nesta estrutura também é utilizada uma tabela de *hash* que mapeia um determinado identificador existente na especificação local de um circuito para um novo identificador que será utilizado na descrição final do circuito. Este renomeação dos identificadores dos fios trata-se de um dos principais problemas a resolver, como tal este referido mapeamento é uma forma de lidar com a necessidade de manter atualizados esses identificadores.

Atualização de identificadores dos fios

De forma a facilitar uma posterior transformação para um descrição num formato mais elementar, no momento que são lidos os *buses* - conjunto de fios - de um determinado circuito, quer sejam de uma *gate* elementar ou de um circuito mais complexos ou de um output, a lista de fios é atualizada tendo em conta o circuito global na qual irão ser inseridos. Um *bus* corresponde a um tuplo de três inteiro:

```
type bus = (gate-id, start-bit, lenght)
```

Este *bus* pode representar um ou mais fios, sendo que após a expansão das *gates* os identificados de *gates* de cada fio terão outro significado que não o pretendido, logo esta **atualização de fios** é relevante pois o circuito geral vai ser composto por uma sequência de sub-circuitos e assim os identificadores dos fios de cada sub-circuito deixam de fazer sentido no contexto geral do circuito. Logo a variável *new_gate_pos* permite um mapeamento do atual identificador para um novo identificador tendo em conta o número de *gates* já lidos até ao momento - que será o identificador no circuito resultante. Assim, esta atualização possi-

bilita que os *buses* já tenham um identificador de acordo com o circuito que será produzido no fim.

Passagem final para um formato elementar

Realizada a expansão total dos sub-circuitos, é possível a transformação em descrições de baixo nível. Para tal é utilizada a seguinte estrutura de dados para captar tal descrição:

```
type elem_circuit = elem array
  and elem = | Input
             | Output of int
             | Gate of (string * int * int option)
```

A representação da informação é um *array* inicializado com o tamanho total do circuito (número total de fios de input + número total de *gates* + número total de fios de output). Nas primeiras posições do *array* estão colocados as instruções (fios) de input. De seguida, para a descrição dos *gates* é percorrida a variável *gates* na estrutura *circuit* adicionando ao *array* o *gate* e respetivos fios de input em caso de ser elementar ou então esta função será invocada recursivamente no caso de ser um sub-circuito. Por fim são adicionados os outputs ao *array*, sendo desdobrados o conjunto de fios de output de forma a que cada posição tenha apenas um fio.

Para explicar o processo realizado podemos observar o exemplo seguinte. Como ficheiro representante do circuito inicial a expandir temos o *3-bitdecode* 4.4 que contém algumas *gates* elementares, mas também duas *gates* com um *decoder* de 2 bits 4.5.

```
0: INPUT 1
1: INPUT 3
2: NOT [(1,2,1)]
3: AND [(0,0,1); (1,2,1)]
4: AND [(0,0,1); (2,0,1)]
5: dec_2 [(4,0,1); (1,0,2)]
6: dec_2 [(3,0,1); (1,0,2)]
7: OUTPUT [(5,0,4); (6,0,4)]
```

Listing 4.4: 3-bit decoder

```
0: INPUT 1
1: INPUT 2
2: NOT [(1,0,1)]
3: NOT [(1,1,1)]
4: AND [(0,0,1); (1,1,1)]
5: AND [(0,0,1); (3,0,1)]
6: AND [(2,0,1); (5,0,1)]
7: AND [(1,0,1); (5,0,1)]
8: AND [(2,0,1); (4,0,1)]
9: AND [(1,0,1); (4,0,1)]
10: OUTPUT [(6,0,1); (7,0,1);
    (8,0,1); (9,0,1)]
```

Listing 4.5: 2-bit decoder

É objetivo da transformação a não existência de *gates* não elementares, logo no caso deste circuito o objetivo será expandir as duas *gates* *dec_2*, mantendo a identificação dos

fios correta de forma a garantir a correção do circuito. Esta expansão será realizada até apenas existirem *gates* elementares. Como o circuito *dec_2* apenas contém *gates* do tipo mais elementar apenas é realizada uma expansão em cada *gate*. Para a passagem para o formato desejado também é necessário expandir os fios inputs de circuito principal, bem como os fios de output. Sendo que o resultado das transformações abordadas seria a descrição em 4.6.

```
0: INPUT
1: INPUT
2: INPUT
3: INPUT
4: NOT 3
5: AND 0 3
6: AND 0 4
7: NOT 1
8: NOT 2
9: AND 6 2
10: AND 6 8
11: AND 7 10
12: AND 1 10
13: AND 7 9
14: AND 1 9
15: NOT 1
16: NOT 2
17: AND 5 2
18: AND 5 16
19: AND 15 18
20: AND 1 18
21: AND 15 17
22: AND 1 17
23: OUTPUT 11
24: OUTPUT 12
25: OUTPUT 13
26: OUTPUT 14
27: OUTPUT 19
28: OUTPUT 20
29: OUTPUT 21
30: OUTPUT 22
```

Listing 4.6: 3-bit decoder em formato elementar

4.2.4 *Funcionalidades adicionais e modo de utilização*

A ferramenta desenvolvida tem diversas funcionalidades que potenciam a capacidade geral da ferramenta.

Uma dessas funcionalidades é a **avaliação da descrição de circuitos**. Esta avaliação é realizada fornecendo um ficheiro de input com um determinado formato. Cada linha corresponde a um conjunto de fios de input, que contém a seguinte descrição: tamanho do input(número de fios) e respetivo valor. Para a representação de *arrays* de inteiros seguido ao tamanho de inputs e apresentado os valores das varias posições do *array*. De notar que esta avaliação tem como alvo apenas operações sob inteiros.

Dado que os circuitos que descrevem as operações a realizar são realizadas e otimizadas à mão é possível que ocorram alguns erros na elaboração das descrições destes circuitos. Assim para contornar tais possíveis erros humanos, no momento da leitura dos circuitos, são realizadas algumas **verificações sob a coerência dessas descrições** que incluem algumas regras como:

- Identificadores de cada linha tem de ser crescentes e com incremento de um de linha para linha;
- No caso de *gates* elementares verificação do número de fios de input é correto para a dada *gate*;
- Verificação se os fios de input na descrição de uma operação de um circuito são coerentes com a descrições dos inputs da descrição do sub-circuito dessa operação;
- Todos os identificadores de fios de input de uma determinada operação têm de ser maiores que o identificador atual da linha.

Como se trata de um ferramenta desenvolvida em OCaml não existe a verificação formal que o Coq garante, logo é necessário que permitir que haja confiança nos resultados que esta produza. Assim foram adicionado alguns **testes unitários** às transformações das descrições, usando o *Kaputt* [2]. Esta ferramenta de testes unitários permite a codificação da especificação da função a testar e permite a geração de valores aleatórios a serem testados face à especificação. Dado isto, para os circuitos que representam operações como adição, multiplicação, seleção e atualização de *arrays*,... foram adicionados testes unitários que cobram estas operações comparando entre a avaliação do circuito que representa a operação e o resultado da operação descrita em OCaml.

Para finalizar também é possível a escrita da descrição do circuito final para um ficheiro, bem como a **geração de estatísticas** sob o circuito. Estas estatísticas contém o número de *gates* totais, bem como o número existente para cada tipo de *gate*.

2 http://kaputt.x9c.fr/

De forma a facilitar o uso da ferramenta nesta sub-secção é descrito a maneira pela qual ser utilizada a ferramenta. A ferramenta é um Command Language Interpreter (CLI) sendo as opções de execução descrita na tabela 2.

Opções de linha de comando	Argumento requerido	Descrição da funcionalidade
-c	Nome do ficheiro que contém a descrição do circuito alvo	Permite a transformação de descrições de circuitos
-o	Nome de um ficheiro novo a criar com o circuito resultante	Permite a escrita num ficheiro do circuito resultante fornecendo o nome desse ficheiro
-i	Nome do ficheiro que contém a informação de input	Permite a avaliação do circuito resultante face ao formato do ficheiro fornecido
-t	-	Gera testes unitários para o circuito, se possivel
-s	-	Gera estatísticas sobre o circuito resultante

Tabela 2: Opções de linha de comando da ferramenta com descrição da funcionalidade e respetivo argumento

4.3 FUNCIONALIDADES A SUPORTAR

O desenvolvimento da ferramenta apresentada na secção 4.2 serviu de protótipo de apoio para a realização da componente final apresentada na secção 3.3.3. Como tal, após implementada a ferramenta foram retiradas diversas conclusões no que diz respeito a funcionalidades que poderiam ter sido elaboradas de forma a melhorar o desempenho da mesma. Os principais aspetos que serão mencionados nesta secção foram implementados na componente equivalente que se encontra no CDG.

A necessidade de melhorar a ferramenta desenvolvida levou a uma análise crítica sobre alguns componente a mudar ou acrescentar permitindo uma melhoria da performance. As principais ilações a retiram serão apresentadas como soluções de problemas ou otimizações que foram implementadas na versão atual do compilador.

Gates parametrizadas

Uma vez elaboradas algumas das descrições de circuitos, foi encontrada a oportunidade de exprimir determinados circuitos de uma forma mais genérica. Ou seja, em vez de especificar n circuitos que executem uma mesma operação, dependendo apenas de um outro

fator, como o tamanho, poderia-se especificar apenas uma *gate* que recebesse um argumento captando as mesmas propriedades dos outros circuitos.

Um exemplo ilustrativo de *gates* deste tipo são os vários circuitos que operam a função de *decode* de dois ou três ou n bits. Assim numa solução futura apenas existiria uma implementação da *gate* de *decode*, onde seria fornecido um novo argumento que corresponderia aos fios de input do tamanho do valor do *decode* a realizar.

Assim, na versão atual do compilador, estão implementadas *gates* parametrizadas que permitem a expressão de múltiplas operações que requerem argumentos como *decode*, seleções ou atualizações em *arrays*.

Eliminação de gates inacessíveis

A melhoria da performance do circuito é um fator determinante no desenvolvimento do compilador. Como tal, foram feitas análises de forma alcançar tal objetivo. As próximas medidas abordadas são apresentadas como funcionalidades que conseguem uma melhoria na avaliação do circuito Booleano.

Uma das medidas óbvias para melhorar a avaliação do circuito é reduzir o tamanho do mesmo, assegurando sempre a manutenção da correção do circuito. Uma das formas de alcançar isto é retirar as *gates* que não estejam a ser utilizadas, método parecido ao *dead-code elimination*. Esta remoção de *gates* poder ser alcançada propagando os fios de output até aos fios de input. Com esta propagação dos fios é possível fazer uma verificação de acessibilidade das *gates*, sendo removidas aquelas que não forem alcançáveis.

Memoization

Ainda dentro da temática de poupança ou remoção de *gates*, outra abordagem que se pode referir é a reutilização do resultado de operações iguais que recebam os mesmos inputs. Para tal é introduzido o conceito seguinte:

A ***Memoization***, introduzido por Michie [1968], é uma técnica de otimização usada essencialmente para melhorar o *speed-up* de programas, guardando o resultado de funções pesadas, retornando o valor guardado dessas operações quando os mesmos inputs ocorrerem outra vez.

Neste contexto, este conceito consiste na memorização de *gates* com certos inputs, utilizando uma estrutura de dados que guarde o tipo de *gate* bem como os fios de input e associando o resultado a um fio. Assim quando for utilizado esse tipo de *gate* com esses inputs, é desnecessário criar uma nova *gate* com essa operação, sendo apenas necessário reutilizar o fio de output guardado.

5

ANÁLISE DO CDG

Nesta capítulo será realizada uma análise dos resultados produzidos pelo compilador CDG, comparando as estatísticas das descrições geradas com outro as estatísticas de outros circuitos Booleanos produzidos por outras ferramentas como o CBMC-CG, abordado na secção 2.4, e os circuitos fornecidos pelo grupo de investigação da Universidade de Bristol, referido na secção 3.3.3. Numa segunda parte da análise realizada foi escolhido um caso de estudo ilustrativo de determinadas de determinados fatores ou conclusões a reter no momento do uso do compilador certificado de circuitos Booleanos.

5.1 ANÁLISE COMPARATIVA

Numa primeira comparação é apresentada a tabela 6 onde se encontram os resultados obtidos para diferentes algoritmos obtidos pelo CBMC-CG e CDG.

Na tabela referida são apresentados os resultados obtidos pela compilação do CBMC-CG e do CDG. O CBMC-CG foi compilado na versão 0.9 apresentada por Franz et al. [2014]. Os algoritmos utilizados foram disponibilizados pelo CBMC-CG, tendo sido adaptados posteriormente para a sua compilação no compilador certificado realizado. Também foram utilizados outros exemplos que se revelaram importantes para esta análise comparativa.

Como as *gates xor* são avaliadas, teoricamente, sem qualquer custo, a medida útil utilizada para analisar os dados de um circuito é o número de *gates* que não sejam *xor*. Juntamente a esta medida também é utilizado o número total de *gates* do circuito. Para uma perceção entre os resultados obtidos é também apresentada a diferença entre tais resultados, sendo que a medida utilizada é a percentagem de número de *gates* ganha ou perdida no CDG em relação ao CBMC-CG.

	Número de gates – CBMC-CG		Número de gates – CDG		Diferença (%)	
	Total	Não Xor	Total	Não Xor	Total	Não Xor
Adição	154	31	154	31	0,0	0,0
Multiplicação	3286	1361	2824	993	-14,1	-27,0
3 * 3 multiplicação de matrizes	85986	32913	79020	27369	-8,1	-16,8
5 * 5 multiplicação de matrizes	398100	151375	368400	127225	-7,5	-16,0
8 * 8 multiplicação de matrizes	1683968	674048	1514880	522304	-10,0	-22,5
Comparação	225	98	191	64	-15,1	-34,7
100 Operações Aritméticas	46211	16168	41855	12657	-9,4	-21,7
1000 Operações Aritméticas	931008	328561	420165	126819	-54,9	-61,4
3000 Operações Aritméticas	1404726	499956	1262665	381568	-10,1	-23,7
Bubble-sort, 11 elementos	17850	12135	17655	5335	-1,1	-56,0
Bubble-sort, 21 elementos	67050	46095	67410	20370	0,5	-55,8
Bubble-sort, 31 elementos	147600	101880	149265	45105	1,1	-55,7
Distancia de Hamming, 160 bit	1610	387	3660	1014	127,3	162,0
Distancia de Hamming, 320 bit	3260	785	7450	2059	128,5	162,3
Distancia de Hamming, 800 bit	8244	1995	18820	5194	128,3	160,4
Distancia de Hamming, 1600 bit	17764	4628	37770	10419	112,6	125,1
AES	35514	35162	32008	7200	-9,9	-79,5
SHA256	114885	32482	118845	28589	3,4	-12,0

Figura 6: Comparação entre o CBMC-CG e o CDG para diferentes algoritmos

Retirando uma primeira impressão dos resultados apresentados na tabela podemos concluir que para grande parte dos algoritmos testados existe um ganho - redução de *gates não xor* - nas descrições de circuitos gerados pelo CDG.

Na medida mais significativa, o número de não *xor*, podemos observar que existe um ganho significativo para quase todos os algoritmos. O fato do ganho ser mais pronunciado nesta medida comparativamente à outra revela o cuidado na implementação do CDG na produção de circuitos com o menor número de *xors* possíveis.

Mesmo em determinados algoritmos em que o número de total de *gates* é ligeira maior, o número de *gates não xor* é substancialmente inferior. A exceção a esta análise é o algoritmo que calcula a distância de *Hamming*. Neste caso tanto o número total como o número de não *xor* sofrem um aumento considerável.

Na tabela 7 é introduzido um novo conjunto de circuitos Booleanos, que foram desenvolvidos por uma equipa de investigação da Universidade de Bristol. As estatísticas destes circuitos foram também confrontados com os resultados do CDG.

Na representação desta tabela é utilizada a medida de número total de *gates*, bem como o número de *gates and*.

	Número de gates – Bristol		Número de gates – CDG		Diferença (%)	
	Total	And	Total	And	Total	And
AES	33616	6800	32008	6400	-4,8	-5,9
SHA256	236112	90825	118845	25667	-49,7	-71,7

Figura 7: Comparação entre os circuitos de Bristol e o CDG para diferentes algoritmos

Tal como na comparação anterior é visível que para os dois algoritmos existem ganhos em relação à redução do número de *gates*.

No exemplo do AES existe um ganho na redução de *gates and*, em que apesar de ser uma ligeiramente melhor traduz um

Esta comparação de resultados entre estes dois algoritmos será alvo de estudo na secção seguinte, onde será dado uma explicação mais detalhada da obtenção dos resultados.

5.2 CASO DE ESTUDO

Nesta secção será realizada uma análise crítica sobre os resultados obtidos pela ferramenta CDG. Esta análise irá recair sobre os resultados alcançados no algoritmo AES e suas variantes, bem como, na função criptográfica Secure Hash Algorithm - 256 (SHA256). A comparação destes resultados será confrontada com os resultados fornecidos pela Universidade de Bristol. Durante esta análise serão estudadas diversas implementações dos algoritmos utilizados de forma a perceber a melhor forma de tirar partido das características do CDG.

O objeto de estudo desta análise é a comparação dos diferentes resultados obtidos nos casos do AES e SHA256. Como vamos poder observar irá ocorrer um diferença entre os resultados obtidos entre

Análise do AES

Para a análise sobre o algoritmo AES foram utilizadas duas implementações distintas, de modo a avaliar o impacto que diferentes estratégias de implementação podem afetar a qualidade dos circuitos obtidos. Cada uma destas versões é apresentada de seguida, sendo ambas *open-source*:

- AES-tab32 - implementação baseada no versão inicial do AES onde nas transformações entre cada ronda são utilizadas tabelas de *lookup* distintas para a cifragem e decifragem;

- AES-sbox - versão mais simplificada onde é utilizado uma *substitution box - s-box* - no passo *SubBytes*, sendo, portanto, as tabelas de *lookup* substituídas por cálculos *'on the fly'*.

Com estas duas variantes foram obtidos os seus resultados relativamente ao número de cada tipo de *gate*, e estes, foram comparados com as estatísticas dos circuitos fornecidos pelo grupo de investigação da Universidade de Bristol. A tabela 8 relaciona o número de *gates and* e *xor* para essas versões, e também é calculado o ganho ou perda percentual relativamente à primeira versão.

	XOR		AND	
	#	%	#	%
AES-tab32	439656	-	50800	-
AES-sbox	259624	-41	50800	0
AES-Bristol	25124	-94	6800	-87

Figura 8: Comparação do número dos vários tipos *gate* para os diversos resultados do AES

Face aos resultados da tabela 8, a primeira grande conclusão que podemos retirar é que os resultados obtidos pelo CDG, para ambas as versões, são significativamente inferiores relativamente aos de Bristol. A enorme diferença, mais de 85 por cento, de *gates and* revela a fraca performance que as descrições geradas obterão. Apesar de existir uma diminuição do número de *gates* da versão tabulada para a versão com *s-box* e da eliminação das múltiplas tabelas existentes na primeira versão, não existe uma diminuição do número de *gates and* na segunda versão.

Identificação e minimização do bottleneck do algoritmo

Para percebemos a razão pela qual existe um número elevado de *gates and* em ambas as versões testadas é necessário identificar os fatores que estão a provocar estes números. Assim, dada a análise anterior, podemos concluir que o número elevado de *gates and* é provocado essencialmente pelo acesso à *s-box*, assim é necessário compreender como é realizado a seleção em *arrays* pelo compilador.

Atualmente no compilador existem duas formas genéricas de realizar consultas em posições de *arrays*. A forma mais simples de se realizar essa consulta é com o recurso à *gate* **selk**. Esta recebe como parâmetros a largura do elemento (w), o tamanho total em bits do *array* (s) e a posição pretendida (p). O circuito que representa esta *gate* recebe apenas um input que corresponde aos bits que representam a informação do *array*, não sendo necessário o recurso a outro tipo de *gates* para representar a lógica da operação, visto que basta o circuito ter como output o conjunto de fios, com tamanho w, a partir da posição $w * p$ do conjunto

de bits *s*. Assim podemos verificar que esta *gate* executa em tempo constante, não sendo necessário um conjunto de *gates* complexas para a representar.

Um outra forma de realizar seleções é através de uma *gate* parametrizada: **sel_W_N**, onde os parâmetros são a largura do elemento e o tamanho total em bits do *array*. Na representação deste circuito é utilizado como input, para além do tamanho em bits do *arrays*, um conjunto de bits que representa a posição desejada. A diferença para a *gate* anterior é que esta não recebe como parâmetro a posição, pois não é possível determinar no momento da compilação esse valor. O valor desta posição é adicionado como input do circuito o que obriga a que seja necessário realizar uma descodificação da posição, bem como, a tradução do valor correspondente dessa posição no *array*. Estas operação impõem a utilização de um conjunto de *gates and* que degrada a performance geral do circuito.

Uma vez identificado o *bottleneck* da geração deste tipo de implementações do AES, é possível tentar evitar as consequências que esta acarreta. Assim, foi utilizado uma nova versão deste algoritmo que emula o acesso à *S-box* num circuito Booleano. Através desta otimização é possível um menor recurso a *gates and* para exprimir a lógica da operação. Nesta nova descrição não existe a lógica de seleção de *arrays* das implementações referidas anteriormente, logo não existe esse tipo de *gates* que provocam o aumento de *gates and*.

	selk	sel_W_N
AES-tab32	36	200
AES-sbox	36	200
AES-opt-sbox	32	0

Tabela 3: Número de cada tipo de *gate* de seleção nas três implementações do AES

Para a elaboração da tabela 3 foram geradas descrições no formato 3.3.2 e retirado dessas descrições o número de cada tipo de *gate* enunciado na figura. Como é percetível pela tabela com a emulação da *s-box* num circuito Booleano foi possível a não utilização das *gates sel_W_N*, que irá provocar uma melhoria da descrição gerada. Assim, na tabela 9 é realizada a comparação entre os resultados obtidos nesta versão com os dados de Bristol.

	AES-Bristol	AES-opt-sbox	Diferença(%)
Nº XOR	25124	24806	-1
Nº AND	6800	6400	-6

Figura 9: Comparação do número dos vários tipos *gate* entre os dois resultados do AES

Pela tabela exposta podemos verificar que existe uma melhoria tanto no número de *gates and* como de *xor*, que apesar de traduzir um ganho percentual reduzido traz melhorias no momento da avaliação do circuito. Para além desta melhoria em relação ao circuito de Bristol, de notar a enorme diferença, como expectável, que esta nova versão trouxe em

relação às duas versões anteriores do AES que continham um número de *gates* de ambos os tipos bastante elevado.

Análise do SHA256

Um outro algoritmo alvo da análise nesta secção é o SHA256. A implementação usada é uma versão *open-source* do mesmo, sendo que apresenta as principais características desta função criptográfica. Analisando o algoritmo tendo em conta os tipos de seleções que provocaram a análise do algoritmo anterior, apenas existem seleções em estrutura de dados em tempo constante - *gates selk*. Os resultados da compilação deste programa são apresentados na tabela 10, bem como os dados correspondentes ao formato de Bristol.

	SHA256-Bristol	SHA256	Diferença(%)
Nº XOR	42029	90256	+54
Nº AND	90825	25667	-72

Figura 10: Comparação do número dos vários tipos *gate* para os diversos resultados do SHA256

De forma bem denotada podemos concluir que, pelos resultados obtidos, existe uma melhor potencial avaliação da descrição gerada comparativamente ao circuito no formato de Bristol. Apesar de existir um aumento de mais de cinquenta por cento no número de *gates xor* na nossa versão, o número de *gates and*, que é o fator que traduz a complexidade de avaliação da descrição, sofre uma diminuição de setenta e dois por cento, traduzindo-se assim num ganho assinalável na performance da avaliação da descrição gerada.

Conclusões

Uma vez estudados estes dois algoritmos e comparados os resultados produzidos pelo CDG em relação aos resultados dos circuitos de Bristol, podemos retirar algumas ilações deste caso de estudo.

Enquanto que na versão do SHA256 com recurso a uma implementação trivial do função criptográfica foi possível obter resultados competitivos em relação aos de Bristol, no caso do AES não se verificou o mesmo. Neste caso foi necessário o recurso a uma implementação bastante específica de forma a tirar proveito das capacidades do compilador.

Esta dificuldade em representar o AES de forma eficiente, nas primeiras versões apresentadas, prende-se no fato das implementações usadas não utilizarem a política do SHA256. A implementação deste assenta sob uma política de tempo constante, contudo o mesmo não se aplica no caso do AES. Ao invés, este faz uso de múltiplas tabelas de *lookup* que

implicam que o programa não execute em tempo constante, obrigando a que as descrições contenham um tipo de *gates* que prejudicam a avaliação do circuito.

De modo a finalizar, podemos concluir que, dado estes dois exemplos, para se poder atingir uma boa ou melhor descrição final gerada pelo compilador, é necessário ter em atenção detalhes na implementação do algoritmo, pois esta irá ser fator preponderante na eficiência da avaliação do circuito gerado. É por isso missão do programador que utilize este compilador estudá-lo de forma a tirar partir das suas principais características.

6

CONCLUSÕES E TRABALHO FUTURO

6.1 CONCLUSÕES

Nesta dissertação de mestrado foi estudado e apresentado um compilador que é capaz de produzir circuitos Booleanos de forma certificada. Não é fácil a elaboração de descrições de circuitos que necessitam de se encontrar num formato muito especifico, logo foi colocada a hipótese de realizar um compilador que execute essa transformação. A necessidade de executar este processo corretamente e de forma confiável foi um dos aspetos mais desafiantes que foram encontrados.

A utilização de alguns passos do compilador CompCert para a transformação inicial do programa C revelou-se útil para alcançar um estado do programa em que a passagem para descrições de circuitos Booleanos fosse mais facilitada.

Um fator que foi tido em conta no contexto da dissertação, foi a realização de um compilador capaz de produzir as descrições com as melhores medidas de eficiência possíveis. A principal medida é a produção de um circuito com menor número de *gates and* possível. Relativamente a esta medida podemos concluir que, dado o esforço por exprimir as operações desejadas com menor número de *gates* deste tipo, as descrições geradas para diversos algoritmos são computacionalmente mais eficientes quando comparadas com outras soluções. As soluções alternativas, como o CBMC-CG e as descrições de Bristol, revelaram-se com uma performance pior do que a nossa solução apresentada, o CDG. Outro fato importante a reter é que, de forma a retirar o máximo das capacidades do compilador, é importante para o programador que utilizar o compilador perceber como é realizado a instanciação de *gates* de modo a retirar a melhor forma de reduzir o número de *gates* não desejadas.

Nesta dissertação é demonstrado que este compilador certificado de circuito é confiável para a geração de descrições de circuitos Booleanos com medidas de eficiência bastante aceitáveis para a realização de futuras avaliações desse circuito.

6.2 TRABALHO FUTURO

Existem ainda alguns possíveis melhoramentos que podem ser explorados e que irão ser abordados de seguida.

Um melhoramento que se poderia ser equacionado no futuro seria a implementação da última fase do compilador, que atualmente se encontra implementado em OCaml, em Coq. Com isto poderíamos alcançar uma geração das descrições abrangidas por um compilador totalmente certificado.

Outro poderia a otimização da especificação dos circuitos presentes atualmente no compilador. Apesar de o compilador apresentar bons níveis em termos de número de *gates and*, este número poderia ser alvo de uma tentativa de redução de forma a melhorar o resultado final da descrição.

Uma outra forma de abstração que poderá ser realizada é a representação sob a forma de circuitos aritméticos. Assim poderia ser gerado, em alternativa aos circuitos Booleanos, descrições de operações aritméticas como adição, multiplicação, subtração, etc.

BIBLIOGRAFIA

D. Beaver, S. Micali, and P. Rogaway. The round complexity of secure protocols. In *Proceedings of the Twenty-second Annual ACM Symposium on Theory of Computing*, STOC '90, pages 503–513, New York, NY, USA, 1990. ACM. ISBN 0-89791-361-2. doi: 10.1145/100216.100287. URL http://doi.acm.org/10.1145/100216.100287.

Coq development team. The coq proof assistant, 2009. URL http://coq.inria.fr/.

Eric Eide and John Regehr. Volatiles are miscompiled, and what to do about it. In *Proceedings of the 8th ACM International Conference on Embedded Software*, EMSOFT '08, pages 255–264, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-468-3. doi: 10.1145/1450058.1450093. URL http://doi.acm.org/10.1145/1450058.1450093.

Shimon Even, Oded Goldreich, and Abraham Lempel. A randomized protocol for signing contracts. *Commun. ACM*, 28(6):637–647, June 1985. ISSN 0001-0782. doi: 10.1145/3812.3818. URL http://doi.acm.org/10.1145/3812.3818.

Martin Franz, Andreas Holzer, Stefan Katzenbeisser, Christian Schallhart, and Helmut Veith. Cbmc-gc: An ansi c compiler for secure two-party computations. In *International Conference on Compiler Construction*, pages 244–249. Springer, 2014.

Andreas Holzer, Martin Franz, Stefan Katzenbeisser, and Helmut Veith. Secure two-party computations in ansi c. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 772–783, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1651-4. doi: 10.1145/2382196.2382278. URL http://doi.acm.org/10.1145/2382196.2382278.

Vladimir Kolesnikov and Thomas Schneider. *Improved Garbled Circuit: Free XOR Gates and Applications*, pages 486–498. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008. ISBN 978-3-540-70583-3. doi: 10.1007/978-3-540-70583-3_40. URL http://dx.doi.org/10.1007/978-3-540-70583-3_40.

X. Leroy, D. Doligez, J. Garrigue, D. Rémy, and J. Vouillon. *The Objective Caml system, documentation and user's manual (release 3. 06)*, 2002.

Xavier Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43 (4):363–446, 2009a. URL http://gallium.inria.fr/~xleroy/publi/compcert-backend.pdf.

Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7): 107–115, 2009b. URL http://gallium.inria.fr/~xleroy/publi/compcert-CACM.pdf.

John McCarthy and James Painter. Mathematical aspects of computer science. In *Proc. of Symposia in Applied Mathematics*, Proc. of Symposia in Applied Mathematics, page 33–41. American Mathematical Society, 1967.

Donald Michie. Memo functions and machine learning. *Nature*, 218(5136):19–22, 1968.

R. Milner and R. Weyrauch. Proving compiler correctness in a mechanical logic. In *Proc. 7th Annual Machine Intelligence Workshop*, Machine Intelligence, page 51–72. Edinburgh University Press, 1972.

Moni Naor, Benny Pinkas, and Reuban Sumner. Privacy preserving auctions and mechanism design. In *Proceedings of the 1st ACM Conference on Electronic Commerce*, EC '99, pages 129–139, New York, NY, USA, 1999. ACM. ISBN 1-58113-176-3. doi: 10.1145/336992.337028. URL http://doi.acm.org/10.1145/336992.337028.

S. Tillich and N. Smart. *Circuits of basic functions suitable for MPC and FHE*. 2015. URL http://www.cs.bris.ac.uk/Research/CryptographySecurity/MPC/.

Heribert Vollmer. *Introduction to Circuit Complexity: A Uniform Approach*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999. ISBN 3540643109.

Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in c compilers. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 283–294, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0663-8. doi: 10.1145/1993498.1993532. URL http://doi.acm.org/10.1145/1993498.1993532.

A. C.-C. Yao. How to generate and exchange secrets. In *In Proceedings of the 27th Annual Symposium on Foundations of Computer Science,*, page 162–167. IEEE Computer Society, 1986.