UNIVERSIDADE DA BEIRA INTERIOR
Engenharia

SoftCheck, uma plataforma de construção de análises estáticas para a segurança de programas, genérica e composicional

(Versão Final Após Defesa)

João Santos Reis

Dissertação para obtenção do grau de Mestre em
Engenharia Informática
(2º ciclo de estudos)

Orientador: Professor Doutor Simão Melo de Sousa

Covilhã, agosto 2018

Dedicatória

Para os meus pais, irmão e avós, com muito amor.

Para ti Duarte, menino do nosso coração.
Eterna saudade.

Agradecimentos

Esta dissertação é o culminar não só deste trabalho, como também deste ciclo de estudos. Como tal, não poderia deixar de tecer alguns agradecimentos.

Em primeiro lugar, agradecer ao meu orientador, Professor Simão Melo de Sousa. Obrigado por todo o acompanhamento e acolhimento no grupo RELEASE, não só durante a realização deste trabalho, mas também durante todo o meu percurso académico desde o momento em que o Prof. João Paulo Fernandes me convidou a fazer parte do Clube de Programação, ainda no meu primeiro ano académico. Aproveito para agradecer ao Prof. João Paulo Fernandes por me ter convidado a fazer parte deste grupo, pois foi neste grupo que descobri as minhas áreas de interesse e um conjunto de alunos e professores formidáveis.

Quero deixar uma mensagem de agradecimento ao Vitor Pereira pela ajuda e disponibilização de código e informação sobre a linguagem `CAO`. Votos de sucesso na tua carreira Vitor!

Quero agradecer à minha namorada, Mariana Flor, por ter sido uma fonte incansável de motivação. Obrigado por toda a ajuda e compreensão nestes últimos anos, especialmente nos momentos de mais trabalho. Quando nem eu acreditava que era capaz, lá estavas tu para me dizer o contrário. Sem ti não teria sido possível superar tudo isto.

Um agradecimento especial a toda à minha família, em especial, aos meus pais, irmão e avó, que sempre me acompanharam durante toda a minha vida. Obrigado por todas as diferentes formas de apoio que me deram e por todos os esforços e sacrifícios que suportaram para que eu pudesse sempre ter todas as condições para o sucesso. Esse sucesso devo-o a vós.

Por fim, agradeço aos meus colegas de mestrado, nomeadamente, ao Luís Horta, Rui Paiva e Eunice Martins, por terem partilhado comigo incontáveis horas diurnas e noturnas de trabalho, diversão e loucura. Trabalhar sem vocês não é a mesma coisa. As melhores felicidades para todos!

Resumo

A implementação de análises estáticas de programas é muitas vezes feita de forma *ad-hoc* ou especializada para um desígnio particular num contexto programático particular. Como tal, a extensão ou adaptação de análises para outros contextos ou linguagens de programação é um processo moroso. Neste trabalho endereçamos este problema apresentando uma plataforma computacional que permite a expressão de análises estáticas para a segurança de programas arbitrariamente complexas e centrada em análises de fluxos de dados. Tirando proveito de um sistema de módulos expressivo, a plataforma `SoftCheck` permite a definição e resolução de análises de uma forma abstrata à linguagem fonte do programa, tornando o processo mais simples e as análises mais facilmente extensíveis e reutilizáveis. Retratamos também um caso de estudo sobre o uso da plataforma para a linguagem de programação `CAO`, uma linguagem dedicada à programação de componentes criptográficas. Neste caso de estudo, mostramos como se obtém um conjunto de análises clássicas de fluxos de dados e uma análise para a segurança, recorrendo simplesmente ao referido mecanismo de instanciação do `SoftCheck`, demonstrando como as caraterísticas desta permitem instanciar estas ou outras análises com esforço reduzido.

Palavras-chave

Análise estática de programas, *Framework* monótona, Análise de fluxo de dados, Modularidade e composição de análises, Análise estática para segurança.

Abstract

Static program analyses are often implemented in an *ad-hoc* manner, being specialised to a specific programming context. As such, the extension or adaption of static analyses to other contexts or programming languages is a task that often requires considerable effort. In this work, we address these problems presenting a computational platform that allows the definition of static program analyses for security with arbitrary complexity and focused on data flow analyses. Benefiting from an expressive module system, `SoftCheck` platform allows the definition and computation of analyses in a manner independent to the source code language of the program, making the process simpler and the analyses more easily extensible e reusable. We also detail a case study about the usage of this platform applied to `CAO` programming language, a *Domain Specific Language* (DSL) for writing cryptographic components. In this case, we demonstrate how to obtain a set of classical static data flow analyses and a security analysis, only resorting to the referred mechanism of `SoftCheck`, demonstrating how its characteristics allow instantiating these and other analyses with reduced effort.

Keywords

Static program analysis, Monotone framework, Data flow analysis, Modularity and composition of analyses, Static program analysis for security.

Conteúdo

Lista de Figuras

Lista de Tabelas

Lista de Acrónimos

CFG Grafo de Controlo de Fluxo

AST Árvore de Sintaxe Abstrata

DSL *Domain Specific Language*

Capítulo 1

Introdução

Atualmente existe *software* em execução em todo o lado. PCs e telemóveis têm-no no seu cerne, carros e aviões são dirigidos por ele, mesmo em lâmpadas, com a proliferação da *Internet of Things*, é possível encontrar *software*. Mas como o *software* é escrito por humanos, este também está sujeito ao erro humano. Estes erros, referidos geralmente como *bugs*, podem ter origem nas várias etapas de desenvolvimento (especificação, *design* e programação) [20] e existem múltiplas categorizações para estes [3, 6]. Uma falha no *software* pode ocorrer devido a um *bug* e pode impedir que um programa funcione como esperado.

As falhas de *software* têm consequências com vários níveis de severidade. Uma falha de *software* pode-se manifestar como algo com impacto mínimo, p.e., um sistema que apresenta uma data num formato diferente. No entanto, também é possível que uma falha se manifeste sob a forma de um *exploit* que permita obter acesso *root* ou executar código arbitrário, comprometendo assim a segurança do sistema [11]. Um sistema crítico que esteja a correr *software* defeituoso suscetível a falhas pode resultar em grandes percas económicas [10] ou até por em risco vidas humanas [14].

Por forma a prevenir falhas de *software*, diferentes ferramentas e técnicas de verificação de *software* são usadas nas várias etapas de desenvolvimento para prevenir ou detetar *bugs* antes de distribuir o *software*. Existem várias propostas para solucionar o problema da verificação de *software*, sendo uma dessas propostas a análise estática de programas. Esta técnica visa fornecer uma aproximação em tempo de compilação dos resultados e comportamentos esperados da execução de um programa. Como tal, a análise estática de programas pode fornecer informação sobre comportamentos maliciosos ou inesperados de um programa, permitindo ao programador que esteja ciente deste comportamento e possa corrigir o *bug* que origina este comportamento anómalo sem a necessidade de executar o programa.

Propostas como a verificação dedutiva permitem verificar propriedades mais precisas e expressivas em comparação com a análise estática. Estas requerem, no entanto, um maior esforço por parte do utilizador para especificar essas propriedades a serem verificadas, p.e., definir e provar invariantes de ciclo. Apesar de a análise estática de programas apresentar este nível de automatização superior, existem ainda tarefas do processo de definição de uma análise que requerem algum esforço por parte do utilizador.

Todas as análises de programas devem ser baseadas na semântica da linguagem do programa a analisar [16], isto é, toda a informação extraída a partir de uma análise pode ser provada correta com respeito à semântica da linguagem de programação. Como tal, partes do processo de definição de uma análise estão inerentemente ligadas a determinada linguagem de programação. Apesar de haver outras partes do processo que são independentes da linguagem de programação

do programa a analisar, nem sempre as ferramentas que permitem definir as análises proporcionam esta camada de abstração. Isto leva a que o processo de definição de uma análise no âmbito de outra linguagem de programação requeira por vezes uma reescrita completa de toda a análise.

1.1 Objetivos

Neste trabalho, é proposto o desenho e implementação de uma *framework* para análise estática para a segurança de programas. Esta *framework* deve permitir que múltiplas análises estáticas sejam executadas sobre os mesmos algoritmos de resolução, suportado por um sistema modular. Esta deverá suportar também que, para o mesmo tipo de análise, instâncias para diferentes linguagens de programação partilhem o mesmo núcleo de definição de análises, reduzindo desta forma parte do esforço de implementação de uma análise.

É proposto também um caso de estudo, focado na implementação de análises estáticas para a segurança para a linguagem de programação `CAO`, uma DSL desenhada para desenvolver *software* criptográfico. Como complemento a este caso estudo, é acrescentada a implementação de análises estáticas para a linguagem de programação `TIP`.

1.2 Estrutura do Documento

O capítulo 2 deste documento apresenta as bases teóricas para este trabalho. Conjuntos, ordens e reticulados são brevemente introduzidos visto serem as estruturas matemáticas na qual a *framework* e os algoritmos de resolução que a apoiam são construídos. Os algoritmos de resolução, por sua vez, são baseados em ponto fixo, conceito que é também discutido. Neste capítulo são também introduzidos os conceitos fundamentais sobre análise estática de programas, assim como o conceito de *framework* monótona e como este se aplica a análises de fluxo de dados. Por fim, é dada uma visão geral sobre o trabalho relacionado no domínio da análise estática de programas. Cada ferramenta e técnica é sumariada e analisada, de forma a que possa ser elaborada uma análise final comparativa deste trabalho.

No capítulo 3 são referidos os detalhes de *design* e implementação da plataforma `SoftCheck`, ponto primário deste trabalho.

É apresentado um caso de estudo da plataforma `SoftCheck` no capítulo 4. São introduzidas as linguagens `CAO` e `TIP` e é detalhada a sintaxe de cada um. É descrito o processo de criação de análises e de instanciação para duas linguagens de programação. Este capítulo termina com uma descrição dos testes realizados à plataforma e quais os resultados obtidos.

Por fim, no capítulo 5 são tecidas algumas conclusões sobre o trabalho efetuado. São revistos os objetivos propostos inicialmente e são enunciadas algumas linhas de trabalho futuro.

Capítulo 2

Conceitos Introdutórios

2.1 Conjuntos, Ordens e Reticulados

O reticulado é uma estrutura matemática indispensável na análise estática de programas. Como tal, os aspetos e definição deste são sumariados nesta secção. Também é introduzido um teorema sobre reticulados que é fulcral para o desenvolvimento deste trabalho.

2.1.1 Conjuntos Parcialmente Ordenados

Seja S um conjunto. Uma ordem parcial em S é uma relação $\sqsubseteq$: $S \times S \to \{true, false\}$ que é, $\forall x, y, z \in S$,

1. reflexiva (i.e. $x \sqsubseteq x$);
2. anti-simétrica (i.e. $x \sqsubseteq y \wedge y \sqsubseteq x \Rightarrow x = y$);
3. transitiva (i.e. $x \sqsubseteq y \wedge y \sqsubseteq z \Rightarrow x \sqsubseteq z$).

Posto isto, um conjunto parcialmente ordenado é um conjunto S equipado com uma ordem parcial $\sqsubseteq$, representado como o par $(S, \sqsubseteq)$ daqui em diante.

Se existe um elemento $y \in S$ tal que $y \subseteq x$ para todo o $x \in S$, então y é o elemento mínimo de S, também designado de $\bot$ (*bottom*). Na existência de um elemento $z \in S$ tal que $x \subseteq z$ para todo o $x \in S$, então z é o elemento máximo de S, designado por $\top$ (*top*).

Seja P um subconjunto de S. Um elemento $x \in P$ é um majorante de S se $\forall s \in S, s \sqsubseteq x$. De forma análoga, um elemento $y \in P$ é um minorante de S se $\forall s \in S, y \sqsubseteq s$. Denotamos o conjunto de todos os majorantes de S por S^u e o conjunto de todos os minorantes de S por S^l.

Se o conjunto S^u tem um elemento mínimo x, então x é designado supremo de S. O elemento máximo de S^l, se existir, é designado de ínfimo de S. Daqui em diante, o supremo e o ínfimo de um conjunto S poderão ser referidos como $\bigsqcup S$ e $\bigsqcap S$, respetivamente. Definimos também os operadores $\sqcup$ (*join*) e $\sqcap$ (*meet*) que representam o supremo e o ínfimo de qualquer subconjunto de S com dois elementos, i.e, seja P um subconjunto de S tal que $x, y \in S$ e $P = \{x, y\}$, escrevemos $x \sqcup y$ para representar o supremo de P e $x \sqcap y$ para representar o ínfimo de P.

2.1.2 Reticulados

Seja $(S, \sqsubseteq)$ um conjunto parcialmente ordenado não-vazio. Se para todo o $x, y \in S$ existe $x \sqcup y$ e $x \sqcap y$, então S é um reticulado. Nos casos em que existe $\bigsqcup P$ e $\bigsqcap P$ para todo o $P \subseteq S$, S é um reticulado completo.

2.1.3 Ponto Fixo

Seja S um conjunto parcialemente ordenado e φ um mapeamento de S para S. Dizemos que um elemento $x \in S$ é um ponto fixo de φ se $\varphi(x) = x$.

2.1.4 Preservação da Ordem em Mapeamentos entre Conjuntos Parcialmente Ordenados

Sejam P e Q dois conjuntos parcialmente ordenados. Diz-se que um mapeamento $\varphi : P \to Q$ preserva a ordem (ou, alternativamente, monótono) se $x \sqsubseteq y$ em P implica que $\varphi(x) \sqsubseteq \varphi(y)$ em Q.

2.1.5 Teorema do Ponto Fixo de Knaster-Tarski

Dado um reticulado completo L, um mapeamento monótono $\varphi : L \to L$, sendo $fix(\varphi)$ o conjunto dos pontos fixos de φ, temos que

$$\alpha = \bigsqcup \{ x \in L \mid x \sqsubseteq \varphi(x) \}, \quad \alpha \in fix(\varphi) \quad \textbf{(2.1)}$$

e

$$\beta = \bigsqcap \{ x \in L \mid \varphi(x) \sqsubseteq x \}, \quad \beta \in fix(\varphi). \quad \textbf{(2.2)}$$

Isto é, dado um reticulado completo L e um mapeamento monótono $\varphi : L \to L$, φ tem necessariamente ponto fixo, sendo que α define o maior ponto fixo de φ e β define o menor ponto fixo de φ. Este teorema é conhecido como o teorema do ponto fixo de Knaster-Tarski [21].

2.2 Análise de Fluxo de Dados

Esta secção introduz o conceito de análise de fluxo de dados. São referidos os vários tipos desta análise e é dado um exemplo de uma análise de fluxo de dados.

2.2.1 Definição

A Análise de Fluxo de Dados, como o nome indica, é uma análise sensível ao fluxo de dados de um programa. Este tipo de análise visa fornecer informação sobre como um programa se comporta extraindo a cada bloco do programa informação relativa ao problema de fluxo de dados particular a ser resolvido, propagando e compondo essa informação ao longo do fluxo do programa. Não é possível obter uma representação real da execução do programa pois é um problema indecidível [19]. No entanto, conseguimos obter uma aproximação do comportamento do programa.

Neste tipo de análises é conveniente representar um programa sob a forma de um grafo dirigido chamado grafo de controlo de fluxo (em inglês, CFG), onde os vértices correspondem aos blocos do programa e as arestas descrevem como o controlo do programa flui de um bloco para o outro. O CFG para o exemplo da figura 4.1 encontra-se na figura 2.1. Para melhor identificação dos blocos do programa, é atribuído uma etiqueta (um número) a cada um.

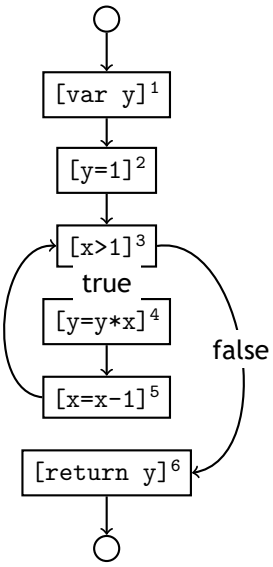


Figura 2.1: CFG para o programa que calcula o fatorial de um núммero.

Uma das abordagens a este tipo de análises consiste em extrair um conjunto de dois tipos de equações. Um desses tipos de equações define a informação à saída de um vértice, o outro tipo de equações define a informação à entrada de um vértice.

Podemos concluir da figura 2.2 que cada equação de fluxo de dados apenas depende das anteriores. No entanto, é possível que existam equações mutuamente recursivas (p.e. quando existem ciclos no programa), o que faz com que não se possa resolver o sistema de equações através de simples substituição.

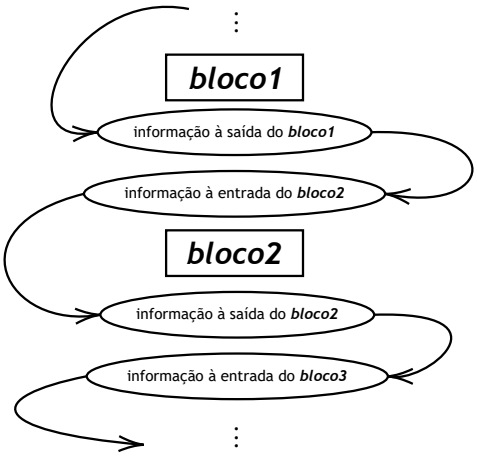


Figura 2.2: Exemplo de dependência das informações à entrada e à saída dos blocos.

2.2.2 Tipos de Análise

Existem classificações das análises de fluxo de dados em função das suas caraterísticas, descritas na tabela 2.1.

Tabela 2.1: Caraterização de análises de fluxo de dados.

Em função da direção	
Para a frente	**Para trás**
Informação computada de acordo com o comportamento anterior	Informação computada de acordo com o comportamento futuro

Em função do tipo de aproximação	
May	***Must***
Descreve informação que pode ser verdadeira (subaproximação)	Descreve informação que tem de ser verdadeira (sobreaproximação)

Ao analisarmos o fluxo de dados de um programa, existem duas variantes relativas ao âmbito do programa, intraprocedimental e interprocedimental. Na variante intraprocedimental analisamos cada função isoladamente. Na variante interprocedimental são tidas em conta as chamadas e retornos de funções assim como as passagens de parâmetros.

2.2.2.1 Análise Interprocedimental

A variante interprocedimental de uma análise de fluxo de dados faz surgir algumas complicações que não existem na variante intraprocedimental. Ao termos em conta as chamadas e retornos de funções, algumas questões e problemas surgem. Nomeadamente, o contexto das chamadas das funções e os apontadores de funções.

Uma função pode ser chamada várias vezes durante a execução de um programa e em diferentes pontos deste. Como tal, conforme o ponto onde é chamada, esta poderá ter comportamentos diferentes. Desta forma, introduzimos o conceito de contexto. Este contexto irá permitir que

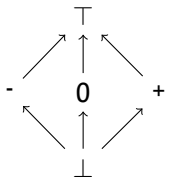
seja possível codificar informação sobre os caminhos percorridos até determinada chamada para dentro das propriedades de fluxo de dados.

Uma das abordagens de implementação deste contexto consiste em defini-lo como um mapeamento de *strings* de chamada para estados das propriedades de fluxo de dados. Desta forma, é possível combinar a informação da saída da função à informação do ponto de chamada correspondente.

Quando uma variável pode denotar uma função, perceber a que função corresponde uma chamada dessa função é uma tarefa não trivial, ao contrário de quando chamamos uma função pelo nome. Existe um outro tipo de análise estática auxiliar para lidar com estas situações que dá pelo nome de análise de controlo de fluxo. A tarefa desta análise é calcular uma aproximação conservativa do fluxo interprocedimental de programas escritos em linguagens com construções que permitem a utilização de apontadores de função.

2.2.3 Exemplo: Análise de Sinal

A análise de sinal procura determinar o sinal de todas as variáveis no decorrer do programa. Desta forma, definimos as propriedades L desta análise como sendo o conjunto de estados abstratos que mapeiam variáveis para o seu possível valor de sinal. Este valor de sinal é representado pelo reticulado da figura 2.3. O valor $\bot$ representa uma variável não inicializada e $\top$ representa o valor para os casos onde não é possível determinizar o valor do sinal.



	⊥	-	0	+	⊤
⊥	⊥	-	0	+	⊤
-	-	-	⊤	⊤	⊤
0	0	⊤	0	⊤	⊤
+	+	⊤	⊤	+	⊤
⊤	⊤	⊤	⊤	⊤	⊤

Figura 2.3: Reticulado de sinal e supremo dos seus elementos.

A função de transferência $f^{sinal} : L \to L$ define que o estado abstrato depois de uma atribuição do tipo $X := E$ é igual ao estado abstrato antes dessa definição, exceto que o valor de sinal de X é igual ao resultado da avaliação da expressão E relativamente ao estado abstrato atual. Definimos a função $eval$ que executa essa avaliação relativamente a um estado abstrato σ como:

$$equal(\sigma, X) = \sigma(X)$$
$$equal(\sigma, I) = sign(I)$$
$$equal(\sigma, E_1 \text{ op } E_2) = op_{eval}(eval(\sigma, E_1), eval(\sigma, E_2))$$

A função $sign$ devolve o valor de sinal de uma constante inteira, e op_{eval} realiza uma avaliação de sinal dado um determinado operador.

2.3 *Framework* Monótona

A *framework* monótona [12] fornece um método genérico de definição e extração de equações de análises de fluxo de dados combinando um reticulado e um espaço de funções monótonas. Nesta secção é dada uma formalização deste conceito e de como se aplica a análises de fluxo de dados.

2.3.1 Definição

Uma *framework* monótona é definida pelo triplo $(L, \sqcup, \mathcal{F})$ onde:

- L é o espaço de propriedades, representado por um reticulado completo cujo operador de supremo é $\sqcup$;
- $\mathcal{F}$ é o espaço de funções monótonas associadas a L, isto é:
  1. todas as funções em $\mathcal{F}$ são monótonas:
  $$\forall x, y \in L, \forall f \in \mathcal{F} : f(x \sqcup y) \sqsupseteq f(x) \sqcup f(y);$$
  2. existe uma função identidade em $\mathcal{F}$:
  $$\forall x \in L, \exists f \in \mathcal{F} : f(x) = x;$$
  3. $\mathcal{F}$ é fechado sobre composição de funções:
  $$f, g \in \mathcal{F} \implies f \circ g \in \mathcal{F}$$

  .

2.3.2 Instância

Para especificar uma análise é necessário instanciar uma *framework* monótona. Esta instância é definida pelo par (G, M) onde:

- G é o CFG do programa a analisar;
- M é um mapeamento que faz corresponder a cada vértice de G uma função em $\mathcal{F}$.

Uma instância da *framework* monótona dá origem a um conjunto de equações do tipo

$$Análise_\circ(v) = \bigsqcup\{Análise_\bullet(v') \mid (v', v) \in F\} \sqcup \iota^v_E$$

$$\text{onde } \iota^v_E = \begin{cases} \iota & \text{se } v \in E \\ \bot & \text{se } v \notin E \end{cases}$$

$$Análise_\bullet(v) = f_v(Análise_\circ(v))$$

onde:

- no caso de análises para a frente, $Análise_\circ(v)$ e $Análise_\bullet(v)$ representam a informação à entrada e à saída do vértice v, respetivamente;
- no caso de análises para trás, $Análise_\bullet(v)$ e $Análise_\circ(v)$ representam a informação à entrada e à saída do vértice v, respetivamente;
- F é o conjunto dos pares de vértices (v', v) de G tais que existe uma aresta que liga v' a v ($v' \to v$);
- E é o conjunto dos vértices iniciais de G;
- ι especifica a informação inicial à entrada dos vértices em E;
- f_v é a função em $\mathcal{F}$ correspondente ao vértice v.

2.3.2.1 Exemplos de Instâncias de *Frameworks*

Na tabela 2.2 pode-se encontrar alguns exemplos de instâncias de *frameworks* monótonas para algumas análises de fluxo de dados clássicas.

Tabela 2.2: Instâncias de quatro análises de fluxo de dados clássicas.

	Expressões Disponíveis	Definições Alcançáveis	Expressões Atarefadas	Vivacidade de Variáveis
L	$\mathcal{P}(\mathbf{AExp})$	$\mathcal{P}(\mathbf{Var} \times \mathbf{Lab})$	$\mathcal{P}(\mathbf{AExp})$	$\mathcal{P}(\mathbf{Var})$
$\sqsubseteq$	$\supseteq$	$\subseteq$	$\supseteq$	$\subseteq$
$\bot$	$\mathbf{AExp}_*$	$\emptyset$	$\mathbf{AExp}_*$	$\emptyset$
ι	$\emptyset$	$\{(x, ?) \mid x \in FV(P)\}$	$\emptyset$	$\emptyset$
	must	*may*	*must*	*may*
$\mathcal{F}$	$\{f : L \to L \mid \exists l_k, l_g : f(l) = (l \setminus l_k) \cup l_g\}$			
f_v	$f_v(l) = (l \setminus kill([B]^v)) \cup gen([B]^v)$ onde $[B]^v \in blocks(P)$			

Nesta tabela, temos que:

- $\mathbf{AExp}$ representa expressões aritméticas e $\mathbf{AExp}_*$ é o conjunto de todas as expressões aritméticas de um programa;

- $\mathbf{Var}$ representa variáveis;
- $\mathbf{Lab}$ representa as etiquetas dos blocos;
- P é o programa a analisar;
- FV é a função que obtém o conjunto de variáveis livres de um programa;
- $blocks$ é a função que devolve o conjunto de blocos de um programa;
- gen e $kill$ são as funções que obtêm as propriedades geradas e mortas num bloco, respetivamente.

2.4 Trabalho Relacionado

Existe uma série de trabalhos na área da análise de estática de programas. Nesta secção são mencionadas alguns trabalhos e contribuições da área. Procuramos realçar as caraterísticas que se enquadrem com os objetivos propostos para este trabalho, os de simplificar e modularizar o processo de implementação de análises estáticas.

Calcagno e Distefano [4] propõe uma ferramenta chamada `Infer`, inicialmente concebida para verificar a segurança de memória de programas escritos na linguagem `C`. A proposta inicial alia análises de forma através de técnicas de bi-abdução [5] a lógica de separação por forma verificar algumas propriedades da memória de um programa. Atualmente desenvolvida internamente na Facebook [8], permite a construção de análises estáticas para várias linguagens através de interpretação abstrata. No entanto, as análises estão restringidas às linguagens suportadas pela plataforma (`C`, `Obj-C`, `C++` e `Java`).

A *framework* de análise dinâmica de *tainting* proposta por Clause e Orso [7] procura endereçar problemas semelhantes aos que tratamos na nossa proposta, apesar de abordar análises dinâmicas ao invés da nossa proposta. Esta *framework* assume a forma da ferramenta `Dytan` que procura permitir a implementação genérica e com esforço reduzido destas análises. Esta *framework* atua sobre executáveis x86.

Mezzetti e Møller [15] desenvolveram uma plataforma de análises estáticas restrita à linguagem de programação `TIP`. Esta plataforma permite definir e executar análises tirando proveito de um conjunto de estruturas e *solvers* pré-existentes. Para além de análises de fluxo de dados, a plataforma fornece análises de ponteiros e uma análise de controlo de fluxo.

Andreas Garnæs [9] produziu uma biblioteca que permite a definição de *frameworks* monótonas em `OCaml`. Produziu também um caso de estudo desta biblioteca definindo algumas análises estáticas para a linguagem `REFRACT` e instanciando as *frameworks* monótonas correspondentes. Esta biblioteca influenciou o trabalho atual na medida em que demonstrou uma implementação de *frameworks* monótonas, neste caso, com suporte a uma única linguagem. A partir daí foi

feito um esforço de repensar toda essa implementação de forma a dar origem a uma plataforma de análise estática com suporte a análises independentes da linguagem.

Capítulo 3

Plataforma `SoftCheck`

Neste capítulo é descrita a nossa abordagem a uma plataforma que permite uma definição e instanciação de análises de fluxo de dados de forma simples e reutilizável. Nomeadamente, descrever como aproveitamos a *framework* monótona para definir um método de resolução de análises genérico e como utilizamos um sistema de módulos e functores por forma a abstrair o processo o mais possível.

O objetivo primário desta proposta é uma plataforma que permita definir e executar análises de fluxo de dados, minimizando o esforço de (re)defini-las para uma ou mais linguagens. Esta plataforma deverá suportar os vários tipos de análises de fluxos de dados mencionados anteriormente (intraprocedimental, interprocedimental, para a frente, para trás, *may*, *must*).

3.1 Arquitetura da Plataforma

A *framework* monótona apresenta um algoritmo genérico para extração de equações de fluxo de dados, o que permite reduzir uma parte do esforço de definição de uma análise. Como tal, a nossa abordagem centra-se em volta desta *framework*. Esta plataforma reflete-se neste trabalho sob a forma da ferramenta `SoftCheck`.

Tendo em conta os objetivos definidos para a plataforma `SoftCheck`, o *design* da mesma teve como foco a divisão do processo de definição de uma análise em três componentes:

1. definição de uma análise;
2. definição de uma instância de uma análise para uma determinada linguagem;
3. resolução da análise.

Esta divisão manifesta-se na plataforma por construção, isto é, a plataforma tenta esclarecer e obrigar metodologicamente a esta divisão. Na figura 3.1 temos uma representação gráfica deste processo e a distinção dos vários componentes. Cada um destes componentes, apesar de interligados, é definido de forma independente dos outros com recurso a abstrações. Isto leva a que tenhamos uma separação de preocupações, como será possível perceber pelos exemplos mais à frente.

O componente específico da análise corresponde a toda a porção de uma análise que é possível definir independentemente da linguagem. Concretamente, neste componente define-se o

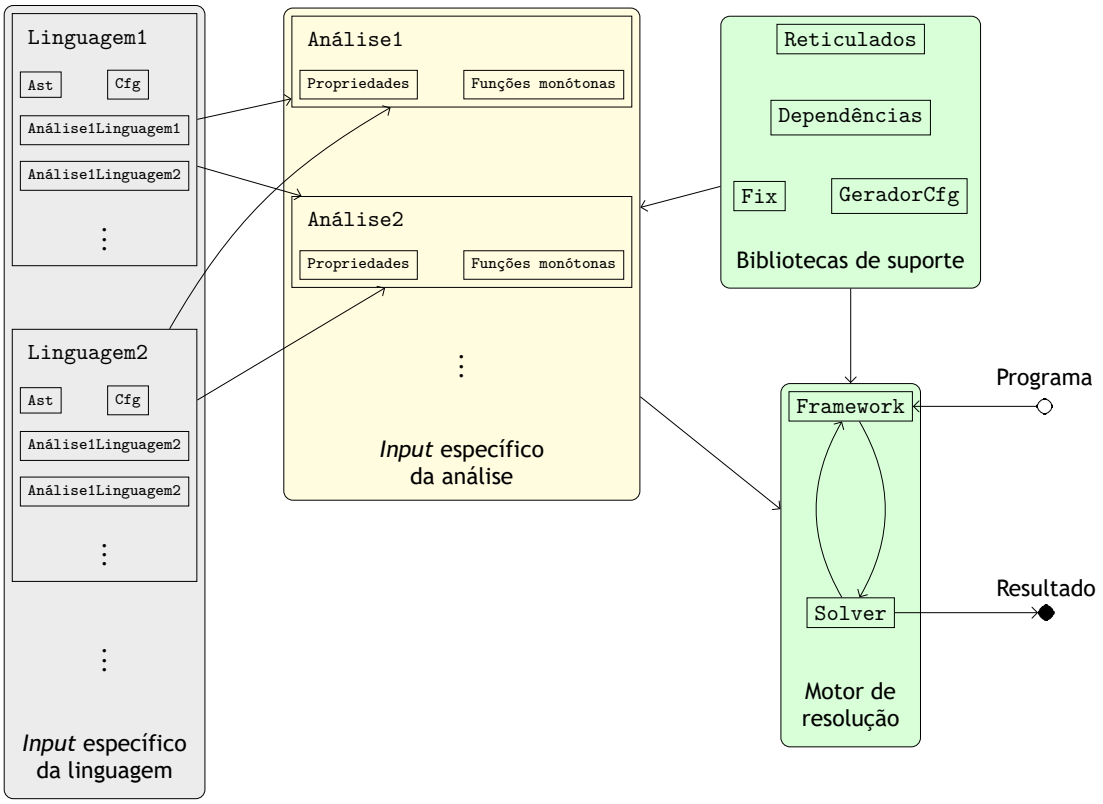


Figura 3.1: Processo de definição e execução de análises na plataforma `SoftCheck`.

espaço de propriedades da análise assim como o espaço de funções de transferência.

O componente específico da linguagem, por sua vez, representa a porção de uma análise que está dependente da semântica da linguagem para a qual se pretende definir essa análise. Este componente engloba também a transformação de um programa para uma representação conhecida pela plataforma e a definição dessa representação (Árvore de Sintaxe Abstrata (AST) e CFG).

A plataforma fornece um motor de resolução de análises, composto pela *framework* monótona e um algoritmo de resolução de sistemas de equações monótonas. Para facilitar a escrita de análises, a plataforma disponibiliza também algumas bibliotecas de suporte que incluem a predefinição das dependências de fluxo para as análises para a frente e para trás e que permitem instanciar diferentes tipos de reticulados completos e gerar o CFG de um programa. O gerador de CFG, para além de criar uma representação em memória dessa estrutura, permite gerar uma visualização do grafo em formato `DOT`.

O componente específico da linguagem alimenta o componente específico da análise, criando desta forma uma instância de uma análise para uma determinada linguagem. Esta instância, por sua vez, irá alimentar o motor de resolução que, provido de um programa a analisar, irá instanciar uma *framework* monótona, gerar o conjunto de equações de fluxo de dados e proceder à sua resolução.

3.1.1 Exemplo

Consideremos um exemplo em que se pretende escrever uma análise de raiz para uma determinada linguagem. O *designer* de análises apenas terá como tarefa a definição da componente específica da análise e da componente específica da linguagem (componentes da figura 3.1 a cinzento e amarelo, respetivamente).

Consideremos agora outro exemplo, em que já existe uma análise definida para uma linguagem `X` e se pretende definir a mesma análise para uma linguagem `Y`. Nesta situação, o *designer* de linguagens não terá que redefinir a componente específica a análise, pois esse componente é definida uma única vez para cada análise. Neste caso, apenas terá de ser definida o componente específico da linguagem `Y` para a análise em questão.

Com os dois exemplos anteriores é possível perceber qual o nível de separação de preocupações que esta plataforma fornece. Este faz com que haja uma redução do esforço de redefinição de uma análise para outros contextos e permite que o utilizador, durante o processo de definição de uma análise, se foque no essencial em cada um dos componentes.

3.2 Detalhes de Implementação da Plataforma

Esta plataforma foi implementada com recurso à linguagem de programação `OCaml`. A escolha desta linguagem baseou-se nos vários pontos fortes da mesma, dos quais queremos destacar, tornando-se claro mais à frente o porquê, o sofisticado sistema de módulos desta linguagem. Este sistema permite organizar os módulos hierarquicamente e parametrizar a instância de um módulo com recurso a um conjunto de outro módulos. Estes módulos parametrizáveis são designados de functores [13].

Em `OCaml`, todos os pedaços de código são envolvidos num módulo. Opcionalmente, um módulo em si pode ser um submódulo de outro módulo. Um módulo pode fornecer tipos, valores (funções são valores em `OCaml`), ou outros módulos para o resto do programa que o usa. Na figura 3.2 temos um exemplo de um módulo `OCaml`.

```
module X = struct
    type t = int

    let initial = 0
    let to_string n = string_of_int n
end
```

Figura 3.2: Exemplo de um módulo `OCaml`.

Assinaturas são interfaces para os módulos. Uma assinatura especifica quais os componentes que um módulo deve ter e estão acessíveis a partir do exterior. As assinaturas em `OCaml` apresentam uma notação semelhante à notação matemática. Um comparativo das duas notações encontra-se na tabela 3.1.

Tabela 3.1: Comparativo da notação matemática com a notação OCaml.

	Matemática	OCaml
Variável inteira x	$x \in \mathbb{Z}$	`val x : int`
Função f que recebe um inteiro e devolve um inteiro	$f : \mathbb{Z} \to \mathbb{Z}$	`val f : int -> int`

Na figura 3.3 temos a assinatura do módulo da figura 3.2.

```
module type Element = sig
    type t

    val initial : t
    val to_string : t -> string
end
```

Figura 3.3: Exemplo de uma assinatura de um módulo OCaml.

Um functor em OCaml é, de grosso modo, uma função de módulos para módulos. Na definição de um functor é necessário especificar as assinaturas de cada um dos módulos que este vai receber. Na figura 3.4 temos um exemplo de um functor que recebe dois módulos, ambos com a assinatura da figura 3.3.

```
module Make_pair(E1 : Element)(E2 : Element) = struct
    type t = E1.t * E2.t

    let initial = E1.initial, E2.initial
    let to_string (e1,e2) =
        Printf.sprintf "%s,%s" (E1.to_string e1) (E2.to_string e2)
end
```

Figura 3.4: Exemplo de um functor OCaml.

Os functores podem ser usados para resolver uma variedade de problemas de estruturação de código, tais como:

- injeção de dependências - torna a implementação de alguns componentes do sistema intercambiáveis;
- autoextensão de módulos - os functores fornecem um modo de estender módulos existentes com novas funcionalidades de uma forma padronizada;
- instanciar módulos com um estado - os módulos podem conter estados mutáveis, o que significa que pode ser desejado ocasionalmente que existam múltiplas instâncias de um módulo em particular, cada uma com o seu estado mutável próprio e independente.

É possível então perceber que estes problemas que os functores vêm resolver são problemas evidentes no contexto deste trabalho, daí a escolha sobre a linguagem de implementação da plataforma tenha recaído sobre esta linguagem.

Assumimos conhecimentos básicos de `OCaml` daqui em diante. Como tal, limitamos os exemplos sob esta assunção. Mais detalhes sobre a linguagem poderão ser encontrados na ligação em [1].

3.2.1 Definição de uma Análise

Como vimos anteriormente, quando definimos o componente específico da análise estamos a definir as propriedades do programa que queremos analisar e o espaço de funções monótonas sobre estas propriedades. Estes são definidos como um functor que é instanciado quando lhes é fornecido componente específico da linguagem. Este functor deverá cria um novo functor a ser instanciado posteriormente quando se pretende executar a análise, sendo-lhe por isso fornecido o programa a analisar. Este functor, por sua vez, define o reticulado completo com o respetivo operador de supremo e o conjunto de funções de transferência, que correspondem ao espaço de propriedades L, ao operador $\sqcup$ e ao espaço de funções monótonas $\mathcal{F}$, respetivamente. Como tal, teremos dois módulos que representam estas estruturas. Na figura 3.5 temos a representação de como um módulo de uma análise deve ser construído.

```
module Make(Ast : Sig.Ast)
    (Printer : Sig.Printer with type expr = Ast.expr)
    (Cfg : Sig.Flow_graph with type stmt_label = Ast.label and
        type program = Ast.program)
    (S : sig ... end) = struct
    module Solve(P : sig val p : Ast.program end) = struct
        module L = ...

        module F = ...

        module Solver = ...

        let solution =  ...

        let get_entry_result l = ...
        let get_exit_result l = ...
        let result_to_string = ...
    end
end
```

Figura 3.5: Estrutura de um módulo de uma análise.

O módulo `S` recebido pelo functor `Make` representa o componente específico da linguagem. Este functor recebe também o módulo `Cfg` com a representação do CFG da linguagem para a qual se pretende instanciar a análise, e o módulo `Printer` com as funções que se encarregam do *prettyprint*.

O functor `Solve` possui os módulos `L` e `F` que definem o reticulado completo das propriedades a analisar e o espaço de funções monótonas sobre essas propriedades, respetivamente. O módulo que define o reticulado completo associado ao espaço de propriedades tem a assinatura `Lattice`. Esta assinatura possui as seguintes definições:

- um tipo `property` que representa o tipo das propriedades;
- o valor `bottom` que corresponde ao elemento mínimo do reticulado;
- a função `equal` que compara a igualdade entre duas propriedades;
- a função `is_maximal` que devolve `true` caso uma propriedade seja um elemento máximo do reticulado;
- a função `lub` que é o operador de supremo;
- a função `to_string` que devolve a `string` que representa uma propriedade.

```
module L = Lattices.Reverse_powerset_lattice(struct
    type t = Ast.expr
    let bottom = aexp_star
    let to_string = Printer.exp_to_string
end)
```

Figura 3.6: Módulo do reticulado completo da análise de expressões disponíveis.

Na figura 3.6 temos um exemplo de uma instância deste módulo para a análise de expressões disponíveis. Este foi criado com recurso a um functor disponível numa biblioteca de reticulados, descrita mais à frente nesta secção.

Temos também a assinatura `Transfer` que define não só o espaço de funções monótonas associado ao espaço de propriedades mas também o mapeamento dos vértices do CFG para as funções monótonas correspondentes, onde:

- o tipo `vertex` coincide com o tipo dos vértices do CFG;
- o tipo `state` reflete o tipo das propriedades;
- o tipo `label` corresponde aos identificadores numéricos de cada vértice;
- o tipo `ident` representa o identificador da função chamada;
- o valor `initial_state` diz respeito ao valor ι das equações geradas;
- a função `f` corresponde ao mapeamento e consequente aplicação da função monótona.

Um exemplo de um módulo com esta assinatura para a análise de expressões disponíveis encontra-se na figura 3.7.

O functor `Solve` cria também uma instância de um *solver* (detalhado mais à frente), assim como as funções que permitem obter o resultado desse *solver* para cada ponto do programa.

Na figura 3.8 temos o módulo que implementa a análise de expressões disponíveis de acordo com as necessidades desta plataforma.

```
module F = struct
    type label = Cfg.stmt_label
    type vertex = Cfg.vertex
    type state = L.property
    type ident = Cfg.ident

    let f _ _ b s =
        let g = S.gen b in
        let k = S.kill aexp_star b in
        (s -. k) ||. g

  let initial_state = Set.empty
end
```

Figura 3.7: Módulo do espaço de funções monótonas da análise de expressões disponíveis.

```
module Make(Ast : Sig.Ast)
    (Printer : Sig.Printer with type expr = Ast.expr)
    (Cfg : Sig.Flow_graph with type stmt_label = Ast.label and type program = Ast.program)
    (S : sig
       val aexp_star : Ast.program -> Ast.expr Set.t
       val gen : Cfg.vertex -> Ast.expr Set.t
       val kill : Ast.expr Set.t -> Cfg.vertex -> Ast.expr Set.t
     end) = struct
  module Solve(P : sig val p : Ast.program end) = struct
    let aexp_star = S.aexp_star P.p

    module L = ...

    module F = ...

    include Analysis.Forward.Make_solution(L)(Cfg)(F)(P)
  end
end
```

Figura 3.8: Implementação da análise de expressões disponíveis.

3.2.1.1 Reticulados do Espaço de Propriedades

Para conveniência do utilizador, a plataforma disponibiliza uma série de functores que permite criar módulos correspondentes a vários tipos de reticulados completos com a assinatura descrita anteriormente. Estes functores são instanciados fornecendo um outro reticulado de base e/ou o tipo das propriedades e outros dados adicionais que sejam necessários, conforme o tipo de reticulado que se está a instanciar. Concretamente, os vários functores de reticulados disponibilizados encontram-se na tabela 3.2.

Uma breve descrição acerca de cada functor é dada a seguir:

- `Uniform_product_lattice` - reticulado cujas propriedades são uma lista de `n` propriedades de outro reticulado;

- `Pair_lattice` - reticulado cujas propriedades são um par composto por propriedades de dois outros reticulados;

Tabela 3.2: Functores de reticulados.

Nome do Reticulado	Módulos Requeridos
`Uniform_product_lattice`	`(L : Sig.Lattice)` `(N : sig val n : int end)`
`Pair_lattice`	`(L1 : Sig.Lattice)` `(L2 : Sig.Lattice)`
`Powerset_lattice`	`(D : Sig.Element)`
`Reverse_powerset_lattice`	`(D : sig include Sig.Element` `val bottom : t Set.t end)`
`Map_lattice`	`(D : sig include Sig.Element` `val bottom_elems : t Set.t end)`
`Flat_lattice`	`(X : Sig.Element)`
`Lattice_with_bottom`	`(L : Sig.Lattice)`
`Lattice_with_top`	`(L : Sig.Lattice)`

- `Powerset_lattice` - reticulado cujas propriedades são conjuntos de propriedades de outro reticulado e cujo operador de supremo é a reunião de conjuntos;

- `Reverse_powerset_lattice` - reticulado cujas propriedades são conjuntos de propriedades de outro reticulado e cujo operador de supremo é a interseção de conjuntos;

- `Map_lattice` - reticulado cujas propriedades são mapeamentos de chaves para propriedades de outro reticulado;

- `Flat_lattice` - reticulado em que a única ordem definida é que o máximo é maior que todas as propriedades de um outro reticulado e o mínimo é menor que todas essas propriedades;

- `Lattice_with_bottom` - reticulado cujas propriedades são as propriedades de um outro reticulado com a adição de um elemento mínimo abstrato;

- `Lattice_with_top` - reticulado cujas propriedades são as propriedades de um outro reticulado com a adição de um elemento máximo abstrato.

3.2.1.2 Gerador de CFG

Foi implementado um functor que gera a representação do CFG de um determinado programa. Este functor gera um módulo, cuja assinatura se encontra na figura 3.9, que define um conjunto de funções importantes para construir e navegar o fluxo de um programa.

As funções `create` e `generate_from_program` permitem criar um novo grafo vazio e gerar um grafo a partir de um programa, respetivamente. As funções `add` e `get` permitem adicionar e remover, respetivamente, um vértice ao grafo. A função `connect` liga dois vértices com uma aresta. As funções `inflow` e `outflow` permitem obter uma lista de vértices que fluem para e de um determinado vértice, respetivamente. As funções `extremal` e `is_extremal` permitem marcar e verificar se determinado vértice é um ponto de entrada do programa, respetivamente. Analogamente, as funções `extremalR` e `is_extremalR` permitem marcar e verificar se determi-

```
module type Flow_graph = sig
  type t
  type program
  type stmt_label = int
  type ident = string
  type vertex
  type edge_label

  val create : unit -> t
  val generate_from_program : program -> t
  val add : t -> string -> (stmt_label * vertex) -> unit
  val get : t -> stmt_label -> vertex
  val connect : t -> ?label:edge_label -> stmt_label -> stmt_label -> unit
  val inflow : t -> stmt_label -> stmt_label list
  val outflow : t -> stmt_label -> stmt_label list
  val extremal : t -> stmt_label -> unit
  val is_extremal : t -> stmt_label -> bool
  val extremalR : t -> stmt_label -> unit
  val is_extremalR : t -> stmt_label -> bool
  val get_blocks : t -> (stmt_label, vertex) Hashtbl.t
  val get_func_id : t -> stmt_label -> ident
  val show : t -> unit
end
```

Figura 3.9: Assinatura do módulo gerado pelo functor gerador de CFG.

nado vértice é um ponto de saída do programa, respetivamente. A função `get_blocks` devolve um mapeamento de todos os vértices de um CFG. A função `get_func_id` permite obter o nome da função à qual pertence um determinado vértice de um CFG. Por fim temos a função `show` que gera a visualização do CFG em formato `DOT`.

3.2.2 Instanciação de uma Análise para uma Determinada Linguagem

Como foi visto anteriormente, ao definirmos uma linguagem estamos a criar um functor que irá receber um módulo com a componente específica da linguagem. Instanciar uma análise para uma determinada linguagem consiste em implementar esse módulo e instanciar o functor. Recordemos o exemplo da figura 3.8. A assinatura do módulo da componente específica (módulo `S`) diz-nos que é necessário implementar nesse módulo três funções. Desta forma, o módulo da componente específica de uma determinada linguagem para esta análise será semelhante ao apresentado na figura 3.10.

3.2.3 Instanciação de uma *Framework*

A instanciação de uma *framework* monótona requer que se forneça o par (G, M) correspondente ao CFG, ao mapeamento de vértices do grafo para as funções monótonas respetivas e às dependências de fluxo. Como o mapeamento foi considerado na definição de um dos módulos referidos anteriormente, a instanciação de uma *framework* requer então apenas o CFG do pro-

```
module S = struct
    let aexp_star p = ...

    let kill aexp_star = ...

    let gen s = ...
end

include AvailableExpressions.Make(Ast)(Printer)(Cfg)(S)
```

Figura 3.10: Estrutura de um módulo de uma componente específica de uma linguagem para a análise de expressões disponíveis.

grama a analisar e as dependências de fluxo, para além do espaço de propriedades e respetivo operador supremo e espaço de funções monótonas que definem a framework.

Instanciar uma *framework* monótona nesta plataforma consiste em fornecer o CFG ao *solver* que irá gerar as equações de fluxo de dados e fornecer estas ao motor de resolução. Como tal, para além dos módulos que definimos anteriormente que representam a definição de uma *framework* (módulos `L` e `F`), são necessários para instanciar uma *framework* outros dois módulos. O módulo do CFG, que deve fornecer todas as funções que permitam obter todos os vértices que fluem de e para determinado vértice, assim como verificar se determinado vértice é um extremo. O módulo das dependências de fluxo deverá definir as funções correspondentes ao tipo de análise (para a frente ou para trás) que permitam obter as dependências de entrada e de saída de um vértice. A plataforma fornece functores que instanciam, a partir do CFG, os módulos de dependências de fluxo correspondentes a cada tipo de análise (para frente ou para trás).

Recordemos o exemplo da figura 3.8. Podemos ver que o *solver* é aqui instanciado recorrendo ao functor `MakeFix`. Na figura 3.11 encontra-se a implementação deste functor. Este define um tipo `variable` que corresponde às equações de fluxo de dados $Análise_\circ$ e $Análise_\bullet$. É também definida a função `generate_equations` que gera as equações de fluxo de dados. Esta função é posteriormente utilizado na função `solve` que recebe o CFG do programa a analisar e gera as equações de fluxo de dados que são fornecidas à biblioteca de resolução.

De notar que as equações geradas são da forma:

$$Análise_\circ(v) = \bigsqcup\{Análise_\bullet(v') \mid (v', v) \in F\} \sqcup \iota_E^v$$

$$\text{onde } \iota_E^v = \begin{cases} \iota & \text{se } v \in E \\ \bot & \text{se } v \notin E \end{cases}$$

$$Análise_\bullet(v) = f_v(Análise_\circ(v))$$

São também definidas as funções que permitem obter o resultado da análise à entrada e à saída de um determinado bloco, assim como a função para obter a *string* correspondente a um determinado resultado.

Foram também implementados functores que facilitam esta tarefa de instanciar uma *framework*,

```
module MakeFix(L : Sig.Lattice)(Cfg : Sig.Flow_graph)
    (F : Sig.Transfer with type label = Cfg.stmt_label
      and type vertex = Cfg.vertex and type state = L.property)
    (D : Sig.Dependencies with type g_t = Cfg.t and
      type label = Cfg.stmt_label)
= struct
  type variable = Circ of Cfg.stmt_label | Bullet of Cfg.stmt_label
  module Fix = Fix.Make(ImperativeMap(struct type key = variable end))(L)

  let generate_equations graph var state =
    match var with
    | Circ l -> D.indep graph l |> List.map (fun l' -> state(Bullet l'))
                |> List.fold_left L.lub
                  (if D.is_extremal graph l then F.initial_state
                   else L.bottom)
    | Bullet l ->
        F.f (Cfg.get_func_id graph l) l (Cfg.get graph l) (state (Circ l))

  let solve graph = generate_equations graph |> Fix.lfp
end
```

Figura 3.11: Functor do *solver*.

um para cada tipo de análise (para frente e para trás). Estes functores instanciam o *solver* com as dependências adequadas e criam as funções para obter os resultados de acordo com o tipo de análise também. É possível comparar a implementação de ambos módulos na figura 3.12.

```
module Solver = Solvers.MakeFix(L)(Cfg)(F)(Dependencies.Forward(Cfg))

let solution = Cfg.generate_from_program P.p |> Solver.solve

let get_entry_result l = solution (Solver.Circ l)
let get_exit_result l = solution (Solver.Bullet l)
let result_to_string = L.to_string
```

(a) Para a frente.

```
module Solver = Solvers.MakeFix(L)(Cfg)(F)(Dependencies.Backward(Cfg))

let solution = Cfg.generate_from_program P.p |> Solver.solve

let get_entry_result l = solution (Solver.Bullet l)
let get_exit_result l = solution (Solver.Circ l)
let result_to_string = L.to_string
```

(b) Para trás.

Figura 3.12: Implementação dos functores que instanciam um *solver*.

3.2.4 Resolução das Equações da Fluxo de Dados e Obtenção de Resultados

Vimos anteriormente que uma *framework* monótona é definida por um reticulado completo e respetivo operador de supremo que representam o espaço de propriedades da análise, designado L, e pelo espaço de funções monótonas $\mathcal{F}$ cujas funções são do tipo $f : L \to L$. Quando estamos

perante estas condições o teorema do ponto fixo de Tarski garante-nos a existência de um ponto fixo para todas as funções do espaço de funções monótonas.

Desta forma, o sistema das equações de fluxo de dados de uma dada instância da *framework* monótona podem ser resolvidas com recurso a algoritmos de ponto fixo. Assim sendo, utilizamos a biblioteca `Fix` [17] de François Pottier, que permite computar a menor solução de um sistema de equações monótonas de forma *lazy*, isto é, apenas são computados os valores das equações à medida que são requisitados.

Esta biblioteca fornece o functor `Make` que recebe dois módulos: uma implementação de um mapa sobre variáveis e um conjunto parcialmente ordenado de propriedades. Este functor produz a função `lfp` que recebe um sistema de equações monótonas e produz a sua menor solução.

A implementação do mapa imperativo deve obedecer à assinatura presente na figura 3.13. O módulo `Hashtbl` fornecido na linguagem `OCaml` providencia construções semelhantes às requeridas que servem o propósito de trabalho. Desta forma, a implementação do mapa imperativo utilizada neste trabalho é definida com recurso a um functor que recebe um módulo com o tipo da chave e produz as funções que operam sobre as tabelas de *hash*, como é possível verificar na figura 3.14.

```
module type IMPERATIVE_MAPS = sig
  type key

  type 'data t

  val create : unit -> 'data t
  val clear : 'data t -> unit
  val add : key -> 'data -> 'data t -> unit
  val find : key -> 'data t -> 'data
  val iter : (key -> 'data -> unit) -> 'data t -> unit
end
```

Figura 3.13: Assinatura dos mapas imperativos.

```
module ImperativeMap
    (K : sig type key end) = struct
  type key = K.key
  type 'data t = (key, 'data) Hashtbl.t

  let create () = Hashtbl.create 10
  let clear ht = Hashtbl.clear ht
  let add k v ht = Hashtbl.replace ht k v
  let find k ht = Hashtbl.find ht k
  let iter f ht = Hashtbl.iter f ht
end
```

Figura 3.14: Functor de mapas imperativos.

A assinatura dos módulos dos conjuntos parcialmente ordenados de propriedades encontra-se na figura 3.15. Esta assinatura apenas requer que seja definido o tipo das propriedades (`property`), o valor mínimo do conjunto parcialmente ordenado (`bottom`) e a função de igualdade entre

propriedades (`equals`). Todos os reticulados que seguem a assinatura `Sig.Lattice` respeitam também esta assinatura, uma vez que a primeira assinatura inclui a segunda. Assim sendo, podemos afirmar que todos os módulos de reticulados referidos anteriormente estão preparados para serem utilizados com esta biblioteca.

```
module type PROPERTY = sig
    type property
    val bottom: property
    val equal: property -> property -> bool
end
```

Figura 3.15: Assinatura dos conjuntos parcialmente ordenados de propriedades.

Capítulo 4

Caso de Estudo

Neste capítulo é retratado um caso de estudo efetuado sobre esta plataforma com o objetivo de demonstrar como simplificámos e tornámos mais reutilizável todo o processo de definição de análises estáticas. Neste sentido, decidimos adotar duas linguagens de programação para as quais implementamos uma série de análises estáticas de programas que são introduzidas também neste capítulo. Por fim, são descritos no fim deste capítulo alguns testes que foram efetuados.

4.1 Linguagem `CAO`

`CAO` [2] é uma DSL para o desenvolvimento de software criptográfico. O foco desta linguagem é fornecer *out-of-the-box* um conjunto de construções matemáticas e criptográficas que permitam uma implementação natural por construção, maximizando a segurança e eficiência dos mesmos através de análises de segurança, transformações e otimizações por parte do compilador. Como tal, apesar de preservar caraterísticas de linguagens imperativas, não suporta construções de input/output e o seu modelo de memória e o seu modelo de memória não contempla alocação dinâmica de memória nem semânticas de *call-by-value*.

A escolha para este caso de estudo recaiu sobre esta linguagem, e não em outras como `C` ou `Java`, pelo facto de esta DSL ter um foco na criptografia, área onde é clara a importância da segurança e fiabilidade. Para além do mais, linguagens como `C` ou `Java` dispõe já de um conjunto alargado de várias ferramentas de análise estática, ao passo que não existe ainda nenhuma ferramenta de análise estática para a linguagem `CAO`. Não obstante, de reforçar que a plataforma desenvolvida no âmbito deste trabalho não é uma plataforma dedicada ao `CAO`, mas sim uma plataforma que procura ser utilizada com toda e qualquer linguagem de programação.

4.1.1 Sintaxe

É apresentada a seguir uma formalização da sintaxe a linguagem `CAO`. Esta é muito semelhante à sintaxe da linguagem de programação `C` e contempla uma série de operadores e estruturas matemáticas que são largamente utilizadas no desenvolvimento de *software* criptográfico.

4.1.1.1 Expressões

e :: L | x | −e | e_1 + e_2 | e_1 − e_2 | e_1 * e_2 | e_1 / e_2 | e_1 % e_2 | e_1 ** e_2 | e.fi | (t) e
| fp(e_1, ... ,e_n) | e_1 == e_2 | e_1 != e_2 | e_1 < e_2 | e_1 > e_2 | e_1 <= e_2 | e_1 >= e_2 | e_1 || e_2
| e_1 && e_2 | !e | e_1 ∧∧ e_2 | e_1[e_2] | e_1[e_2..e_3] | e_1[e_2,e_3] | e_1[e_2,e_3..e_4] | e_1[e_2..e_3,e_4]
| e_1[e_2..e_3,e_4..e_5] | ~e | e_1 & e_2 | e_1 ∧ e_2 | e_1 | e_2 | e_1 « e_2 | e_1 » e_2 | e_1 <| e_2 | e_1 |> e_2
| e_1 @ e_2

4.1.1.2 Literais

l :: x | l[e] | l[e_1..e_2] | l[e_1,e_2] | l[e_1,e_2..e_3] | l[e_1..e_2,e_3] | l[e_1..e_2,e_3..e_4] | l.fi

4.1.1.3 Ciclos

c :: dv | l_1, ... , l_n := e_1, ... ,e_m; | return e_1, ... ,e_n; | fp(e_1, ... ,e_n); | if (e) { c_1; ... ; c_n }
| if (e) { c_{11}; ... ; c_{1n} } else { c_{21}; ... ; c_{2m} } | while (e) { c_1; ... ; c_n }
| seq x := e_1 to e_2 by e_3 c_1; ... ; c_n | seq x := e_1 to e_2 { c_1; ... ; c_n }

4.1.1.4 Declaração de Tipos

dt :: typedef tid := t; | typedef sid := struct [def fi_1 : t_1; ... ; def fin : t_n;];

4.1.1.5 Declaração de Variáveis

dv :: def x : t; | def x_1, ... , x_n : t; | def x : t := e; | def x : t := e_1, ... ,e_n ;

4.1.1.6 Declaração de Funções

dfp :: def fp (x_1 : t_1, ... , xn : t_n) : rt c_1; ... ; cm

4.1.1.7 Tipos de Retorno

rt :: void | t_1, ... , t_n

4.1.1.8 Tipos

t :: int | bool | unsigned bits [e] | signed bits [e] | mod [e] | mod [t/pol] | vector [e] of t
| matrix [e_1,e_2] of t | tid | sid

4.1.1.9 Programa

pg :: dv | dt | dfp | pg_1 pg_2

4.2 Linguagem TIP

A linguagem de programação TIP [15] visa apresentar uma sintaxe minimal (*à la* C), fornecendo ainda assim todas as construções relevantes para a aplicação dos diferentes conceitos de análise estática. Adotámos também esta linguagem porque, apesar desta ser uma linguagem para fins demonstrativos, esta apresenta estas construções que não estão presentes na linguagem CAO e outras (como alocação dinâmica de memória, apontadores de funções e interações de input/output), vindo desta forma complementar este caso de estudo.

4.2.1 Sintaxe

É apresentada a seguir a sintaxe da linguagem TIP com recurso a gramáticas livres de contexto.

4.2.1.1 Expressões

As expressões TIP resumem-se a operações de inteiros, operações de ponteiros e chamadas de função.

```
I :: 0 | 1 | -1 | 2 | -2 | ...
X :: x | y | z | ...
E :: I
 | X
 | E + E | E - E | E * E | E / E | E > E | E == E
 | (E)
 | input
 | X(E, ..., E)
 | &X
 | alloc
 | *E
```

| `null`
| (E)(E, ..., E)

As expressões `TIP` incluem constantes inteiras (I) e variáveis (X). Existe também a expressão `input` que lê um inteiro da *input stream*. Os operadores de comparação produzem 0 para condições falsas e 1 para condições verdadeiras. A linguagem permite também chamadas de função diretas e computadas (apontadores de funções), alocação dinâmica de memória com recurso à expressão `alloc` que aloca uma nova célula de memória na *heap*, criação de ponteiros para variáveis e o `dereferencing` desses ponteiros.

4.2.1.2 Instruções

As instruções `TIP` são compostas pelas estruturas de controlo `if`, `else` e `while`, a instrução `output` que escreve um inteiro para a `output stream`, atribuição de variáveis e composição de instruções.

S :: X = E;
| `output` E;
| S S
|
| `if` (E) { S } `else` { S }
| `if` (E) { S }
| *X = E

4.2.1.3 Funções

A declaração de uma função `TIP` contém o nome da função, um conjunto de parâmetros, a declaração de variáveis locais, o corpo e a instrução `return`.

F :: X (X, ..., X) $[$`var` X, ..., X;$]^{?}$ S `return` E;

Um exemplo de uma declaração de uma função `TIP` encontra-se na figura 4.1.

```
factorial(x) {
    var y;
    y = 1;
    while (x > 1) {
        y = y * x;
        x = x - 1;
    }
    return y;
}
```

Figura 4.1: Função que calcula o fatorial de um determinado inteiro.

4.2.1.4 Programas

Um programa não é mais que uma sequência de funções. A função com o nome `main` é considerada o ponto de entrada do programa.

P :: F ... F

4.3 Análises Implementadas

Nesta secção damos uma descrição das várias análises implementadas neste caso de estudo, com destaque para a análise de *tainting*.

4.3.1 Análises de Fluxo de Dados

Foram implementadas algumas análises estáticas de fluxo de dados clássicas. Para auxiliar o processo de análises de fluxo de dados, nomeadamente na sua vertente interprocedimental, foi implementado também uma análise de controlo de fluxo. Esta análise permite obter uma aproximação conservativa do CFG de um programa quando neste existem estruturas como apontadores de funções.

4.3.1.1 Expressões Disponíveis

A análise de expressões disponíveis procura indicar, num determinado ponto de execução de um programa, que expressões já foram computadas anteriormente e que não foram alteradas desde então. Na vertente conservativa desta análise apenas se pretende obter as expressões que foram computadas e não foram alteradas em nenhum dos caminhos do programa para esse ponto.

4.3.1.2 Vivacidade de Variáveis

A análise de vivacidade de variáveis consiste em verificar se uma variável num determinado ponto do programa é lida posteriormente sem ser escrita entretanto. Esta análise segue uma adoção conservativa no sentido em que as variáveis "não vivas" são sempre verdadeiros negativos mas as variáveis vivas podem ser falsos positivos.

4.3.1.3 Definições Alcançáveis

As definições alcançáveis de um determinado ponto do programa são as definições que podem influenciar o valor atual das variáveis neste ponto.

4.3.1.4 Expressões Atarefadas

É considerada uma expressão atarefada todas as expressões que são computadas antes de terem o seu valor alterado. A aproximação conservativa desta análise resulta em que apenas as verdadeiras expressões atarefadas são consideradas.

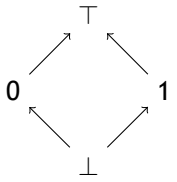
4.3.1.5 Sinal

A análise de sinal foi descrita anteriormente na secção 2.2.3.

4.3.2 Análise de *Tainting*

Uma análise de *tainting* é uma análise onde procuramos perceber se determinada informação que tem origem em fontes de *tainting* (*sources*) atinge pontos sensíveis do programa (*sinks*), introduzindo vulnerabilidades, *leaks* de informação, entre outros. Esta análise é um caso específico de uma análise de fluxo de informação. Um exemplo de uma vulnerabilidade que pode ser detetada é quando uma *string* introduzida pelo utilizador é utilizada numa *query* `SQL` sem qualquer tratamento (`SQL` *injection*).

Esta análise também pode ser expressada com recurso a uma *framework* monótona. Definimos as propriedades desta análise como sendo um mapeamento de variáveis para o seu valor de *tainting*, valor este representado por um reticulado de *tainting* (figura 4.2), com os seguintes estados: indefinido ($\bot$), *tainted* (1), *untainted* (0) e ambos ($\top$).



$\sqcup$	$\bot$	0	1	$\top$
$\bot$	$\bot$	0	1	$\top$
0	0	0	$\top$	$\top$
1	1	$\top$	1	$\top$
$\top$	$\top$	$\top$	$\top$	$\top$

Figura 4.2: Reticulado de *tainting* e operador $\sqcup$.

A análise de *tainting* considera que informação *tainted* atinge uma *sink* quando existe alguma expressão utilizada por essa *sink* marcada como possivelmente *tainted* à entrada desse bloco. Consideramos que existe propagação de informação *tainted* quando no lado direito de uma atribuição existe uma *source* ou outra variável marcada como possivelmente *tainted*.

Esta análise pode ser estendida recorrendo a outra análise mencionada anteriormente, a análise de definições alcançáveis. Esta extensão faz com que a análise não só detete que informação *tainted* pode atingir uma *sink* mas que saiba também quais são os pontos do programa de onde pode originar essa informação.

4.3.3 Definição do Componente Específico das Análises

O processo de definição do componente específico da análise inicia com a criação do módulo do reticulado de propriedades. Com recurso à biblioteca de functores, foi criado o reticulado de *tainting* a partir do functor `Flat_lattice`, onde os elementos são do tipo `bool`, onde `false` significa *untainted* e `true` significa *tainted*. Como pretendemos associar um estado de *tainting* a cada variável do programa, definimos um novo reticulado recorrendo ao functor `Map_lattice`, mapeando identificadores para reticulados de *tainting*. A implementação destes módulos encontra-se na figura 4.3.

```
module Taint_lattice = Lattices.Flat_lattice(struct
    type t = bool
    let to_string = string_of_bool
  end)

module Var_tainting_lattice = Lattices.Map_lattice(struct
        type t = Ast.ident
        let to_string = identity
        let bottom_elems = vars
      end)(Taint_lattice)
```

Figura 4.3: Implementação do reticulado de estados de *tainting*.

Visto que pretendemos utilizar esta análise em junção com a análise de definições alcançáveis, e sendo ambas do mesmo tipo (para a frente), é possível efetuar as duas análises simultaneamente. Como tal, definimos um outro reticulado que representa as propriedades de ambas as análises. Foi criado então o reticulado de propriedades para a análise de *tainting*, definido através do functor `Pair_lattice` que forma um reticulado de pares de outros dois reticulados, o das propriedades de definições alcançáveis e o dos estados de *tainting*. Estas implementações podem ser consultadas na figura 4.4.

```
module Reaching_definitions_lattice = Lattices.Powerset_lattice(struct
    type t = S.definition_location
    let to_string (v,l) = Printf.sprintf "(%s,%s)" v (match l with
        None    -> "?"
        | Some l' -> string_of_int l')
    end)

module L = Lattices.Pair_lattice(Reaching_definitions_lattice)
    (Var_tainting_lattice)
```

Figura 4.4: Implementação dos reticulados das propriedades das análises de definições alcançáveis e de *tainting*.

A implementação dos reticulados das outras análises de fluxo de dados seguem um processo

semelhante, como se pode verificar na figura 4.5.

```
module L =
  Lattices.Reverse_powerset_lattice(
      struct
        type t = Ast.expr
        let bottom = aexp_star
        let to_string =
          Printer.exp_to_string
  end)
```

(a) Expressões disponíveis.

```
module L = Lattices.Map_lattice(
    struct
      type t = Ast.ident
      let to_string = identity
      let bottom_elems =
        declaredVars
    end)(Sign_lattice)
```

(c) Sinal

```
module L =
  Lattices.Powerset_lattice(struct
    type t = Ast.ident
    let to_string = identity
  end)
```

(b) Vivacidade de variáveis.

```
module L =
  Lattices.Reverse_powerset_lattice(
    struct
      type t = Ast.expr
      let bottom = aexp_star
      let to_string =
        Printer.exp_to_string
  end)
```

(d) Expressões atarefadas.

Figura 4.5: Implementação dos reticulados das propriedades das análises de fluxo de dados.

Estando definidas as propriedades das análises, segue-se a implementação a definição do módulo das funções monótonas. Na parte referente à análise de definições alcançáveis, é verificado quais as definições geradas e mortas no bloco e são retiradas do estado atual as mortas e adicionadas as geradas e é definido o estado inicial como o conjunto de pares de cada variável do programa com "?". Na parte referente à análise de *tainting*, é efetuada uma avaliação e são obtidos novos estados de *tainting* gerados no bloco e são introduzidos no mapa. A implementação deste módulo pode ser consultada na figura 4.6.

Os módulos das funções monótonas das restantes análises de fluxo de dados também são implementadas de forma semelhante. As análises de expressões disponíveis, vivacidade de variáveis e expressões atarefadas, todas partilham a mesma estrutura de código com a análise de definições alcançáveis. A análise de sinal, por usa vez, partilha a mesma estrutura com a análise de *tainting*. A implementação deste módulos encontra-se na figura 4.7.

A implementação do componente específico da linguagem fica completa com a adição da instanciação do *solver* e a definição das funções que permitem obter o resultado da análise a cada ponto do programa. Isto é feito através dos functores referidos anteriormente criados para o efeito. No caso da análise de *tainting*, foi adicionada a linha de código presente na figura 4.8.

É possível então perceber que todas as análises seguem a mesma assinatura. Essa mesma assinatura encontra-se na figura 4.9. Também é possível perceber que todas as análises dependem de um módulo `S`. Este módulo corresponde ao componente específico da linguagem, cuja implementação será abordada a seguir.

É possível concluir então que o processo de definição do componente específico da linguagem corresponde de certa forma à definição formal da análise, isto é, das propriedades que essa

```
module F = struct
    type label = Cfg.stmt_label
    type vertex = Cfg.vertex
    type state = L.property
    type ident = Cfg.ident

    let f _ l b s =
        let g = S.gen l b in
        let k = S.kill blocks b in
        let s1 = (fst s -. k) ||. g in
        let s2 = S.ta (snd s) b in
        s1, s2

    let initial_state =
        Set.map (fun x -> x,None) vars,
        Set.fold (fun x acc -> Map.add x Taint_lattice.bottom acc)
            vars Map.empty
end
```

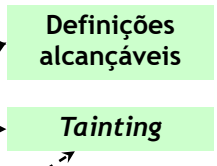


Figura 4.6: Implementação do módulo de funções monótonas das análises de definições alcançáveis e de *tainting*.

```
module F = struct
  let f _ _ b s =
    let g = S.gen b in
    let k = S.kill aexp_star b in
    (s -. k) ||. g

  let initial_state = Set.empty
end
```

(a) Expressões disponíveis, vivacidade de variáveis e expressões atarefadas.

```
module F = struct
  let f _ _ b s =
    let evals = S.sign_eval s b in
    List.fold_left (fun m (i,eval) ->
      L.set m i eval) s evals

  let initial_state = Set.fold (fun x map ->
    Map.add x Sign_lattice.Bottom map)
    declaredVars Map.empty
end
```

(b) Sinal.

Figura 4.7: Implementação dos módulos de funções monótonas das análises de fluxo de dados.

análise pretende avaliar e como é que feito o processo de obtenção de resultados. O processo torna-se assim mais modular, e será evidente mais à frente a forma como a plataforma torna o processo de criação de análises estáticas mais simples.

4.3.4 Definição do Componente Específico da Linguagem

Dada uma análise existente, o processo de adição de suporte a uma nova linguagem a essa análise resume-se aos passos explicados a seguir.

De acordo com a implementação anterior, qualquer análise requer a existência da representação de um programa sob a forma de uma AST e de um CFG. Foram então implementados os módulos que representam as AST das linguagens `CAO` e `TIP` com a assinatura `Sig.Ast`, assim como os módulos dos respetivos *parsers* e *lexers* que permitem obter a AST de um determinado programa.

```
include Analysis.Forward.Make_solution(L)(Cfg)(F)(P)
```

Figura 4.8: Inclusão do *solver* e funções de obtenção de resultados.

```
module Make(Ast : Sig.Ast)
(Cfg : Sig.Flow_graph with type stmt_label = Ast.label and
type program = Ast.program)
(S : sig ... end) : sig
  module Make_solution(P : val p : Ast.program end) : sig
    module L : Sig.Lattice
    module F : Sig.Transfer
    module Solver : Solver.S

    val solution : Solver.Fix.valuation
    val entry : Cfg.stmt_label -> Solver.Fix.property
    val exit : Cfg.stmt_label -> Solver.Fix.property
  end
end
```

Figura 4.9: Assinatura dos módulos de análises.

Tendo estas representações de um programa, segue-se a implementação dos módulos respetivos ao componente específico da linguagem para cada uma das análises.

4.3.4.1 Expressões Disponíveis

Para a análise de expressões disponíveis implementaram-se as funções com as assinaturas presentes a figura 4.10.

```
val aexp_star : Ast.program -> Ast.expr Set.t
val gen : Cfg.vertex -> Ast.expr Set.t
val kill : Ast.expr Set.t -> Cfg.vertex -> Ast.expr Set.t
```

Figura 4.10: Assinatura do módulo específico da linguagem da análise de expressões disponíveis.

A função `aexp_star` devolve o conjunto de todas as expressões aritméticas presentes num programa. A função `gen` devolve o conjunto de expressões aritméticas que passaram a estar disponíveis num determinado ponto do programa. Por sua vez, a função `kill` devolve o conjunto de expressões aritméticas que deixaram de estar disponíveis.

4.3.4.2 Vivacidade de Variáveis

Para a análise de vivacidade de variáveis foram implementadas as funções `gen` e `kill` cujas assinaturas estão presentes na figura 4.11.

A função `gen` devolve o conjunto de todas as variáveis livres que apareçam numa instrução num determinado ponto do programa. A função `kill` devolve o conjunto das variáveis no lado

```
val gen : Cfg.vertex -> Ast.ident Set.t
val kill : Cfg.vertex -> Ast.ident Set.t
```

Figura 4.11: Assinatura do módulo específico da linguagem da análise de vivacidade de variáveis.

esquerdo de uma atribuição, caso seja essa a instrução em causa.

4.3.4.3 Expressões Atarefadas

Na análise de expressões atarefadas implementou-se as funções `aexp_star` e `kill`, ambas com a mesma definição das funções homónimas definidas na análise de expressão disponíveis, assim como a função `gen` que devolve o conjunto de expressões computadas nesse ponto do programa. As assinaturas destas funções encontram-se na figura 4.12.

```
val aexp_star : Ast.program -> Ast.expr Set.t
val gen : Cfg.vertex -> Ast.expr Set.t
val kill : Ast.expr Set.t -> Cfg.vertex -> Ast.expr Set.t
```

Figura 4.12: Assinatura do módulo específico da linguagem da análise de expressões atarefadas.

4.3.4.4 Sinal

A análise de sinal procura determinar o sinal de todas as variáveis no decorrer do programa. Assim sendo, implementou-se a função `declared_vars` que obtém o conjunto de todas as variáveis declaradas num programa. Foi também implementada a função `sign_eval` que computa novos estados de sinal gerados num determinado ponto do programa. As assinaturas destas funções podem ser verificadas na figura 4.13.

```
val declaredVars : Ast.program -> Ast.ident Set.t
val sign_eval : (Cfg.ident, Sign_lattice.property) Map.t ->
  Cfg.vertex -> (Cfg.ident * Sign_lattice.property) list
```

Figura 4.13: Assinatura do módulo específico da linguagem da análise de sinal.

4.3.4.5 Definições Alcançáveis e *Tainting*

A implementação do componente específico da linguagem para a análise de definições alcançáveis requer a definição de dois tipos, `blocks` e `definition_location`. O tipo `blocks` representa uma tabela de hash que mapeia todos os pontos de um programa para o vértice correspondente do CFG. O tipo `definition_location` corresponde às propriedades da análise. Foram também implementadas três funções cujas assinaturas se encontram na figura 4.14.

A função `free_variables` devolve o conjunto de todas as variáveis livres de um programa. As funções `gen` e `kill` obtêm o conjunto de pares de variáveis e localizações de definições gerados

```
type blocks = (Cfg.stmt_label, Cfg.vertex) Hashtbl.t
type definition_location = Cfg.ident * Cfg.stmt_label option
val free_variables : blocks -> Ast.ident Set.t
val gen : Cfg.stmt_label -> Cfg.vertex -> definition_location Set.t
val kill : blocks -> Cfg.vertex -> definition_location Set.t
```

Figura 4.14: Assinatura do módulo específico da linguagem da análise de definições alcançáveis.

e destruídos, respetivamente, num ponto do programa.

Relativamente à análise de *tainting*, a sua implementação consiste em definir a função que compute para cada bloco do programa a informação de *tainting* associada a cada variável. Isto é, sempre que num determinado bloco estamos perante uma atribuição, calcular o valor de *tainting* da variável do lado esquerdo de acordo com a avaliação do valor de *tainting* da expressão do lado direito da atribuição. Definimos essa função com a assinatura da figura 4.15.

```
val ta : (Cfg.ident -> Taint_lattice.property) Map.t -> Cfg.vertex ->
  (Cfg.ident * Taint_lattice.property) list
```

Figura 4.15: Assinatura do módulo específico da linguagem da análise de *tainting*.

A implementação desta função, para a linguagem `CAO` encontra-se na figura 4.16. São tratadas todas as formas de atribuição e gerados os novos valores de *tainting* de cada uma. Foi também implementada uma função auxiliar de avaliação de expressões (figura 4.17. Nesta função de avaliação foi considerado que existe uma função chamada `read` como sendo uma fonte de *tainting*.

```
let ta s b = let open Flow in match b with
    VDecl (VarD v) -> (match v.Ast.var_d_init with
        Some rv -> let eval_rv = eval s rv in
          List.map (fun lv -> (lv, eval_rv)) (get_idents b)
      | None -> List.map (fun lv -> (lv, Taint_lattice.bottom)) (get_idents b))
  | VDecl (ContD c) -> List.map2 (fun lv rv -> (lv, eval s rv))
      c.Ast.cont_d_ids c.Ast.cont_d_init
  | Assign a -> List.map2 (fun lv rv -> let lv2 = get_id_from_lv lv in
      (lv2, eval s rv)) a.Ast.assign_ids a.Ast.assign_values
  | Sample lvals ->
      List.map (fun lv -> (get_id_from_lv lv, Taint_lattice.bottom)) lvals
  | Seq seq -> let eval_s = eval s seq.Ast.seqheader_start_val in
      let eval_e = eval s seq.Ast.seqheader_end_val in
      let eval_t = (match seq.Ast.seqheader_increase with
          Some e ->
            Taint_lattice.lub (Taint_lattice.lub eval_s eval_e) (eval s e)
        | None -> Taint_lattice.lub eval_s eval_e) in
          [seq.Ast.seqheader_var, eval_t]
  | CDecl _
  | FCallS _
  | Ret _
  | Ite _
  | While _ -> []
```

Figura 4.16: Implementação da função `ta` da análise de *tainting* para a linguagem `CAO`.

```
let rec eval s = let open Ast in
  let open Taint_lattice in function
      Var v -> Map.find v s
    | Lit _ -> Taint_lattice.Element false
    | FunCall fc when fc.Ast.funcall_id = "read" -> Element true
    | FunCall fc ->
        List.fold_left (fun acc arg -> join_taint acc (eval s arg))
            bottom fc.Ast.funcall_args
    | StructProj sp -> eval s sp.Ast.structproj_id
    | UnaryOp uop -> eval s uop.Ast.unaryop_expr
    | Access a -> eval s a.Ast.access_container_id
    | Cast c -> eval s c.Ast.cast_expr
    | BinaryOp bop ->
        let eval_e1 = eval s bop.Ast.binaryop_l_expr in
        let eval_e2 = eval s bop.Ast.binaryop_r_expr in
        join_taint eval_e1 eval_e2
```

Figura 4.17: Implementação da função `eval` da análise de *tainting* para a linguagem `CAO`.

É possível perceber então que a adição de uma nova linguagem a uma análise existente se resume à definição de um número reduzido de funções de tratamento de valores nessa linguagem. Assim sendo, começa a evidenciar-se a forma como a plataforma simplifica todo o processo.

4.4 Execução de Análises

Anteriormente definimos de uma forma simples e modular análises estáticas de fluxo de dados. Estas análises foram posteriormente instanciadas para as linguagens `CAO` e `TIPs`, num processo que requereu a escrita de duas a três funções por análise. Desta instanciação resulta um novo functor, que por sua vez cria uma instância de uma solução dessa análise sobre um determinado programa fornecido. Desta forma, o processo de execução de uma análise sobre determinado programa consiste em prover o functor da instância da análise para essa linguagem com o módulo que contém o programa a ser analisado. O módulo gerado por este último functor dispõe dos métodos `get_result_entry` e `get_result_exit` que permitem obter os resultados da análise à entrada e à saída de determinado ponto do programa. Os resultados estão então prontos a serem tratados da forma que o utilizador pretender.

Retomando o caso da análise de *tainting*. Após a execução desta, é feita uma pesquisa no programa para identificar a localização de *sinks*. Tendo adquirido essas localizações, verifica-se que variáveis são usadas por essa *sink* e avalia-se o estado de *tainting* à entrada desse bloco para cada uma dessas variáveis. Caso alguma esteja avaliada como *tainted*, é emitido um aviso que informação *tainted* atinge uma *sink*.

4.5 Testes e Exemplos

Por forma a testar o trabalho realizado, foram criados vários programas para ambas as linguagens a serem analisados.

Foram feitas estas análises no papel e também foram considerados alguns exemplos encontrados na literatura. Os resultados apresentados pela plataforma foram comparados com os resultados no papel, obtendo correspondência em todos os testes.

Na figura 4.18 temos um exemplo de um programa escrito na linguagem `CAO` juntamente com os resultados obtidos pela plataforma para algumas das análises implementadas. Para cada instrução, são apresentadas as propriedades da análise em questão naquele ponto do programa. WW

```
   def Main() : int {
1:   def a, b, x, y : int;
2:   x := a + b;
3:   y := a * b;
4:   while (y > a + b) {
5:     a := a + 1;
6:     x := a + b;
     }
7:   return x;
   }
```

```
Available Expressions
1: {}
2: {(a)+(b)}
3: {(a)+(b),(a)*(b)}
4: {(a)+(b)}
5: {}
6: {(a)+(b)}
7: {(a)+(b)}
```

```
Live Variables
1: {a,b}
2: {a,b}
3: {a,b,y}
4: {a,b,y}
5: {a,b,y}
6: {a,b,x,y}
7: {}
```

```
Very Busy Expressions
1: {(a)+(b),(a)+(1),
    (a)*(b)}
2: {(a)+(1),(a)*(b)}
3: {(a)+(1)}
4: {(a)+(1)}
5: {(a)+(b)}
6: {}
7: {}
```

```
Reaching Definitions
1: {(a,?),(b,?),(x,?),(y,?)}
2: {(a,?),(b,?),(x,2),(y,?)}
3: {(a,?),(b,?),(x,2),(y,3)}
4: {(a,?),(a,5),(b,?),(x,2),
    (x,6),(y,3)}
5: {(a,5),(b,?),(x,2),(x,6),
    (y,3)}
6: {(a,5),(b,?),(x,6),(y,3)}
7: {(a,5),(b,?),(x,6),(y,3)}
```

Figura 4.18: Exemplo de execução de análises de fluxo de dados.

Na figura 4.19 temos um outro exemplo, neste caso de uma análise de *tainting* para a linguagem `TIP`. Consideramos a expressão `input` como uma *source* e a instrução `output` como *sink*. Na instrução 1 é atribuído à variável `n` um valor inserido pelo utilizador, marcando essa variável como *tainted*. Na instrução 3 é atribuído à variável `m` o valor `n + 3`. Como o valor de `n` se encontra num estado *tainted*, este é propagado também para a variável `m`. Na instrução 4 temos então uma situação em que um valor *tainted* (`m`) atinge uma *source* (`output`). Na instrução 8 temos um caso em que informação *tainted* poderá atingir uma *source*. A variável `v` é definida ou na instrução 6 ou 7, pertencentes aos dois ramos de uma instrução de controlo `if`. No caso em que a condição do `if` seja verdadeira, então será atribuído o valor de `m` a `v`, ficando esta com um estado *tainted*. No caso em que a condição se revela falsa, é atribuído o valor 2 à variável `v`. Desta forma, dizemos que informação *tainted* poderá atingir uma *source* porque no caso em que a condição do `if` seja falsa a variável `v` não terá informação *tainted*. O resultado indicado pela plataforma é exatamente este, como é possível verificar na figura.

```
   main() {                  Taint Analysis
     var n, m, v;            Tainted information reaches sink in 4
1:   n = input;              Tainted information may reach sink in 8
2:   m = 2;
3:   m = n + 3;
4:   output m;
5:   if (n > 2) {
6:     v = n;
     }
     else {
7:     v = 2;
     }
8:   output v;
9:   return 0;
   }
```

Figura 4.19: Exemplo de execução de uma análise de *tainting*.

Por questões de espaço, os exemplos aqui apresentamos são simples e de tamanho reduzido, como tal, não devem ser considerados como testes exaustivos. É possível consultar a implementação completa da análise de *tainting* para a linguagem `CAO` no anexo A.1. Como exemplo de implementações mais complexas testadas, foram implementadas algumas construções criptográficas (p.e. `AES`, `SHA1`) para a linguagem `CAO`, as quais se encontram em anexo neste documento (anexo A.2), juntamente com o resultado da análise de expressões disponíveis para estas implementações.

Capítulo 5

Conclusões e Trabalho Futuro

Apresentámos uma proposta de uma plataforma de definição, instanciação e execução de análises estáticas que vem simplificar todos estes processos. Abstraindo algumas das etapas do processo, permitimos que certos componentes da definição de uma análise não necessitem de ser reimplementados quando se quer adotar essa análise num contexto de outra linguagem de programação, tornando assim o processo mais modular e reutilizável.

Concretizámos esta plataforma sob a forma da ferramenta `Softcheck`, cujo código encontra-se na ligação em [18]. Foi exposto um caso de estudo em que, com recurso a esta plataforma, implementámos e executámos várias análises estáticas para as linguagens de programação `CAO` e `TIP`. Neste caso de estudo mostrámos o nível de redução do esforço de implementação de análises estáticas explicitando quais os componentes que definem uma análise e são únicos e reutilizáveis nos vários contextos e quais os componentes cuja implementação é necessária para cada contexto linguístico em que uma análise deseja ser aplicada.

Definimos algumas linhas de trabalho futuro. A inclusão de outras *frameworks* e *solvers* permitirá o suporte a um conjunto mais alargado de tipos de análise. Pretendemos também melhorar a interface da plataforma para uma mais direta implementação em sistemas de integração contínua. Um estudo de como se comporta a plataforma quando aplicada a projetos de grande dimensão é também desejado.

Bibliografia

[1] Ocaml website, https://ocaml.org/ 17

[2] Barbosa, M., Moss, A., Page, D., Rodrigues, N.F., Silva, P.F.: Type Checking Cryptography Implementations. In: Fundamentals of Software Engineering, pp. 316–334. Springer, Berlin, Heidelberg (2012), http://link.springer.com/10.1007/978-3-642-29320-7_21 27

[3] Beizer, B., Boris: Software testing techniques. Van Nostrand Reinhold, 2nd ed. edn. (1990), https://dl.acm.org/citation.cfm?id=79060 1

[4] Calcagno, C., Distefano, D.: Infer: An Automatic Program Verifier for Memory Safety of C Programs. In: Proceedings of the Third International Conference on NASA Formal Methods, pp. 459–465. Springer (2011) 10

[5] Calcagno, C., Distefano, D., O'Hearn, P.W., Yang, H.: Compositional Shape Analysis by Means of Bi-Abduction. Journal of the ACM 58(6), 1–66 (dec 2011) 10

[6] Chillarege, R., Bhandari, I., Chaar, J., Halliday, M., Moebus, D., Ray, B., Wong, M.Y.: Orthogonal defect classification-a concept for in-process measurements. IEEE Transactions on Software Engineering 18(11), 943–956 (1992), http://ieeexplore.ieee.org/document/177364/ 1

[7] Clause, J., Li, W., Orso, A.: Dytan: a generic dynamic taint analysis framework. In: Proceedings of the 2007 international symposium on Software testing and analysis. p. 196 (2007) 10

[8] Facebook: Infer static analyzer, http://fbinfer.com/ 10

[9] Garnæs, A.: Library for specifying monotone frameworks in ocaml, https://github.com/andreas/monotone-framework 10

[10] Gleick, J.: A Bug and a Crash by James Gleick (1996), https://around.com/ariane.html 1

[11] Jalote, P.: List of Common Bugs and Programming Practices to avoid them (2005), https://www.iiitd.edu.in/{~}jalote/papers/CommonBugs.pdf 1

[12] Kam, J.B., Ullman, J.D.: Monotone data flow analysis frameworks. Acta Informatica 7(3), 305–317 (1977) 8

[13] Leroy, X.: Applicative functors and fully transparent higher-order modules. In: Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 142–153. POPL '95, ACM (1995) 15

[14] Leveson, N.: Medical Devices: The Therac-25. In: Safeware: system safety and computers (1995), http://sunnyday.mit.edu/papers/therac.pdf 1

[15] Møller, A., Schwartzbach, M.I.: Static program analysis (September 2017), department of Computer Science, Aarhus University, http://cs.au.dk/~amoeller/spa/ 10, 29

[16] Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Springer-Verlag Berlin Heidelberg (1999) 1

[17] Pottier, F.: Lazy least fixed points in ML (Dec 2009), http://gallium.inria.fr/~fpottier/publis/fpottier-fix.pdf, unpublished 24

[18] Reis, J., Melo de Sousa, S.: Softcheck - github, https://github.com/joaosreis/softcheck 43

[19] Rice, H.G.: Classes of recursively enumerable sets and their decision problems. Transactions of the American Mathematical Society 74(2), 358–358 (feb 1953), http://www.ams.org/jourcgi/jour-getitem?pii=S0002-9947-1953-0053041-6 5

[20] Smith, D.J., Wood, K.B.: Engineering Quality Software : a Review of Current Practices, Standards and Guidelines including New Methods and Development Tools. Springer Netherlands (1989) 1

[21] Tarski, A.: A lattice-theoretical fixpoint theorem and its applications. Pacific Journal of Mathematics 5(2), 285–309 (1955) 4

Apêndice A

Anexos

A.1 Implementação da Análise de *Tainting*

A.1.1 Componente Específico da Análise

```
1  open Batteries
2  open Set.Infix
3  open Utils
4
5  module Make(Ast : Sig.Ast)
6      (Cfg : Sig.Flow_graph with type stmt_label = Ast.label and
7      type program = Ast.program)
8      (S : sig include ReachingDefinitions.Language_component
9         val ta : (ident, Taint_lattice.property) Map.t -> vertex ->
10          (ident * Taint_lattice.property) list
11     end with type ident = Cfg.ident and
12     type stmt_label = Cfg.stmt_label and
13     type vertex = Cfg.vertex) = struct
14   module Solve(P : sig val p : Ast.program end) = struct
15     let graph = Cfg.generate_from_program P.p
16     let blocks = Cfg.get_blocks graph
17     let vars = S.free_variables blocks
18
19     (* Reticulado que mapeia variáveis para estados de tainting *)
20     module Var_tainting_lattice = Lattices.Map_lattice(struct
21         type t = Ast.ident
22         let to_string = identity
23         let bottom_elems = vars
24       end)(Taint_lattice)
25
26
27     (* Reticulado de definições alcancáveis *)
28     module Reaching_definitions_lattice = Lattices.Powerset_lattice(struct
29         type t = S.definition_location
30         let to_string (v,l) = Printf.sprintf "(%s,%s)" v (match l with
31             None    -> "?"
32           | Some l' -> string_of_int l')
33       end)
```

```
34
35      (* Reticulado do par (definições alcançáveis, tainting) *)
36      module L = Lattices.Pair_lattice(Reaching_definitions_lattice)
37          (Var_tainting_lattice)
38
39      module F = struct
40        type label = Cfg.stmt_label
41        type vertex = Cfg.vertex
42        type state = L.property
43        type ident = Cfg.ident
44
45        let f _ l b s =
46          (* Definições alcançáveis *)
47          let g = S.gen l b in
48          let k = S.kill blocks b in
49          let s1 = (fst s -. k) ||. g in
50          (* Tainting *)
51          let new_tv = S.ta (snd s) b in
52          let s2 = List.fold_left (fun m (i,eval) ->
53            Var_tainting_lattice.set m i eval) (snd s) new_tv in
54          s1, s2
55
56        let initial_state =
57          (* Definições alcançáveis *)
58          Set.map (fun x -> x,None) vars,
59          (* Tainting *)
60          Set.fold (fun x acc ->
61            Map.add x Taint_lattice.bottom acc) vars Map.empty
62      end
63
64      include Analysis.Forward.Make_solution(L)(Cfg)(F)(P)
65    end
66  end
```

A.1.2 Componente Específico da Linguagem

```
1  include Taint.Make(Ast)(Cfg)(struct
2      include ReachingDefinitionsCao.S
3
4      (* Função de avaliação do valor de tainting de expressões *)
5      let rec eval s = let open Ast in
6        let open Taint_lattice in function
7            Var v -> Map.find v s
8          | Lit _ -> Taint_lattice.Element false
9          | FunCall fc when fc.Ast.funcall_id = "read" -> Element true
10         | FunCall fc ->
```

```
            List.fold_left (fun acc arg -> join_taint acc (eval s arg))
              bottom fc.Ast.funcall_args
        | StructProj sp -> eval s sp.Ast.structproj_id
        | UnaryOp uop -> eval s uop.Ast.unaryop_expr
        | Access a -> eval s a.Ast.access_container_id
        | Cast c -> eval s c.Ast.cast_expr
        | BinaryOp bop ->
            let eval_e1 = eval s bop.Ast.binaryop_l_expr in
            let eval_e2 = eval s bop.Ast.binaryop_r_expr in
            join_taint eval_e1 eval_e2

    (* Função que obtém novos valores de taiting gerados num determinado
       bloco do programa *)
    let ta s b = let open Flow in match b with
      VDecl (VarD v) -> (match v.Ast.var_d_init with
        Some rv -> let eval_rv = eval s rv in
          List.map (fun lv -> (lv, eval_rv)) (get_idents b)
      | None -> List.map (fun lv -> (lv, Taint_lattice.bottom)) (get_idents b))
    | VDecl (ContD c) -> List.map2 (fun lv rv -> (lv, eval s rv))
                           c.Ast.cont_d_ids c.Ast.cont_d_init
    | Assign a -> List.map2 (fun lv rv -> let lv2 = Ast.get_id_from_lv lv in
                               (lv2, eval s rv)) a.Ast.assign_ids
                    a.Ast.assign_values
    | Sample lvals ->
        List.map (fun lv ->
          (Ast.get_id_from_lv lv, Taint_lattice.bottom)) lvals
    | Seq seq -> let eval_s = eval s seq.Ast.seqheader_start_val in
        let eval_e = eval s seq.Ast.seqheader_end_val in
        let eval_t = (match seq.Ast.seqheader_increase with
            Some e ->
              Taint_lattice.lub (Taint_lattice.lub eval_s eval_e) (eval s e)
          | None -> Taint_lattice.lub eval_s eval_e) in
        [seq.Ast.seqheader_var, eval_t]
    | CDecl _ | FCallS _ | Ret _ | Ite _ | While _ -> []
  end)
```

A.2 Testes

A.2.1 AES

A.2.1.1 Implementação

```
typedef GF2  := mod[ 2 ];
typedef GF2N := mod[ GF2<X> / X**8 + X**4 + X**3 + X + 1 ];
typedef GF2V := vector[8] of GF2;

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

typedef Row  := matrix[1,4] of GF2N;
typedef RowV := vector[4] of GF2N;
typedef Col  := matrix[4,1] of GF2N;
typedef ColV := vector[4] of GF2N;

typedef Byte := unsigned bits[8];
typedef Bit  := unsigned bits[1];

def M : matrix[8,8] of GF2 := { [1], [0], [0], [0], [1], [1], [1], [1],
                                [1], [1], [0], [0], [0], [1], [1], [1],
                                [1], [1], [1], [0], [0], [0], [1], [1],
                                [1], [1], [1], [1], [0], [0], [0], [1],
                                [1], [1], [1], [1], [1], [0], [0], [0],
                                [0], [1], [1], [1], [1], [1], [0], [0],
                                [0], [0], [1], [1], [1], [1], [1], [0],
                                [0], [0], [0], [1], [1], [1], [1], [1] };

def C : vector[8]   of GF2 := { [1], [1], [0], [0], [0], [1], [1], [0] };

def SBox( e : GF2N ) : GF2N
{
    def x : GF2N;
    def A : matrix[8,1] of GF2;
    def B : GF2V;
    if (e == GF2N:0) { x := [0]; }
    else { x := [1] / e; }

    A := matrix[8,1] of GF2:(GF2V:x);
    B := GF2V:(M*A);
    return (GF2N:B) + (GF2N:C);
}

```

```
40
41  def SubBytes( s : S ) : S
42  {
43      def r : S;
44      seq i := 0 to 1 {
45          seq j := 0 to 3 {
46              r[i,j] := SBox( s[i,j] );
47          }
48      }
49      return r;
50  }
51
52  def SubWord( w : vector[4] of GF2N ) : vector[4] of GF2N
53  {
54      def r : vector[4] of GF2N;
55      seq i := 0 to 3 {
56          r[i] := SBox( w[i] );
57      }
58      return r;
59  }
60
61  def ShiftRows( s : S ) : S
62  {
63      def r : S;
64      seq i := 0 to 3 {
65          r[i,0..3] := Row:((RowV:s[i,0..3]) |> i);
66      }
67      return r;
68  }
69
70  def mix : matrix[4,4] of GF2N  :=
71  {
72      [X],   [X+1], [1],   [1],
73      [1],   [X],   [X+1], [1],
74      [1],   [1],   [X],   [X+1],
75      [X+1], [1],   [1],   [X]
76  };
77
78  def MixColumns( s : S ) : S
79  {
80      def r : S;
81      seq i := 0 to 3 {
82          r[0..3,i] := mix * s[0..3,i];
83      }
84      return r;
85  }
86
```

```
87  def AddRoundKey( s : S, k : K ) : S
88  {
89      def r : S;
90
91      seq i := 0 to 3 {
92          seq j := 0 to 3 {
93              r[i,j] := s[i,j] + k[i,j];
94          }
95      }
96
97      return r;
98  }
99
100 def FullRound( s : S, k : K ) : S
101 {
102     return MixColumns( ShiftRows( SubBytes(s) ) ) + k;
103 }
104
105 def makeRCon() : vector[11] of vector[4] of GF2N
106 {
107     def Rcon : vector[11] of vector[4] of GF2N;
108     def lsw : vector[3] of GF2N := { [0],[0],[0] };
109     def t : vector[1] of GF2N;
110
111     seq i := 1 to 10 {
112         t[0] := [X];
113         t[0]:=t[0]**(i-1);
114         Rcon[i] := t @ lsw;
115     }
116
117     return Rcon;
118 }
119
120 def ExpandKey( k : K ) : vector[11] of K
121 {
122     def r : vector[11] of K;
123     def Rcon : vector[11] of vector[4] of GF2N;
124     Rcon := makeRCon();
125     def lsw : vector[3] of GF2N := { [0], [0], [0] };
126     def oldr : K;
127     def curr : K;
128
129     r[0] := k;
130     seq i := 1 to 10 {
131         oldr := r[i-1];
132
133         curr[0..3,0] := (Col:SubWord( (ColV:oldr[0..3,3]) |> 1)) +
```

```
134                             (Col:Rcon[i]) + oldr[0..3,0];
135
136         seq j := 1 to 3 {
137             curr[0..3,j] := curr[0..3,j-1] + oldr[0..3,j];
138         }
139         r[i] := curr;
140     }
141     return r;
142 }
143
144 def Aes( s : S, k : K) : S
145 {
146     def r : S := s+k;
147     def keys : vector[11] of K;
148     keys := ExpandKey(k);
149
150     seq i := 1 to 9 {
151         r := FullRound( r,keys[i] );
152     }
153
154     return ShiftRows( SubBytes(r) ) + keys[10];
155 }
156
157 def encodeByte( x : Byte ) : GF2N
158 {
159     def t : vector[8] of GF2 := { GF2:x[0], GF2:x[1], GF2:x[2], GF2:x[3],
160                                   GF2:x[4], GF2:x[5], GF2:x[6], GF2:x[7] };
161     return GF2N:t;
162 }
163
164 def encodeRow( x : vector[4] of Byte ) : vector[4] of GF2N
165 {
166     def t : vector[4] of GF2N;
167     seq i := 0 to 3 {
168         t[i] := encodeByte( x[i] );
169     }
170     return t;
171 }
172
173 def encodeKey( inp : vector[16] of Byte ) : K
174 {
175     def r : K;
176     seq i := 0 to 3 {
177         seq j := 0 to 3 {
178             r[i,j] := encodeByte( inp[i+4*j] );
179         }
180     }
```

```
181     return r;
182 }
183 
184 def decodeByte( x : GF2N ) : Byte
185 {
186     def t : vector[8] of GF2;
187     t := vector[8] of GF2:x;
188     def r : Byte := (Bit:(int:t[0])) @ (Bit:(int:t[1])) @ (Bit:(int:t[2])) @
189         (Bit:(int:t[3])) @ (Bit:(int:t[4])) @ (Bit:(int:t[5])) @
190         (Bit:(int:t[6])) @ (Bit:(int:t[7]));
191     return r;
192 }
193 
194 def decodeRow( x : matrix[1,4] of GF2N ) : vector[4] of Byte
195 {
196     def r : vector[4] of Byte;
197     seq i := 0 to 3 {
198         r[i] := decodeByte( x[0,i] );
199     }
200     return r;
201 }
202 
203 def decodeKey( x : K ) : vector[16] of Byte
204 {
205     def r : vector[16] of Byte;
206     seq i := 0 to 3 {
207         seq j := 0 to 3 {
208             r[i*4+j] := decodeByte(x[j,i]);
209         }
210     }
211     return r;
212 }
```

A.2.1.2 Resultado da Análise de Expressões Disponíveis

```
1:  {}
2:  {}
3:  {}
4:  {}
5:  {}
6:  {}
7:  {}
8:  {}
9:  {}
10: {}
11: {}
12: {}
```

```
13: {}
14: {}
15: {}
16: {}
17: {}
18: {}
19: {(1)/(e)}
20: {}
21: {}
22: {((GF2N) B)+((GF2N) C)}
24: {}
25: {}
26: {}
27: {}
28: {}
30: {}
31: {}
32: {}
33: {}
35: {}
36: {}
37: {}
38: {}
41: {}
42: {}
43: {(mix)*(s[0..3,i])}
44: {}
46: {}
47: {}
48: {}
49: {(s[i,j])+(k[i,j])}
50: {}
52: {(MixColumns(ShiftRows(SubBytes(s))))+(k)}
54: {}
55: {}
56: {}
57: {}
58: {}
59: {(t[0])**((i)-(1))}
60: {(t[0])**((i)-(1))}
61: {}
63: {}
64: {}
65: {}
66: {}
67: {}
68: {}
```

```
69: {}
70: {}
71: {}
72: {((((Col) SubWord(((ColV) oldr[0..3,3])|>(1)))+((Col) Rcon[i]))+(oldr[0..3,0])}
73: {((((Col) SubWord(((ColV) oldr[0..3,3])|>(1)))+((Col) Rcon[i]))+(oldr[0..3,0])}
74: {((((Col) SubWord(((ColV) oldr[0..3,3])|>(1)))+((Col) Rcon[i]))+(oldr[0..3,0]),
     (curr[0..3,(j)-(1)])+(oldr[0..3,j])}
75: {((((Col) SubWord(((ColV) oldr[0..3,3])|>(1)))+((Col) Rcon[i]))+(oldr[0..3,0])}
76: {}
78: {(s)+(k)}
79: {(s)+(k)}
80: {(s)+(k)}
81: {(s)+(k)}
82: {(s)+(k)}
83: {(s)+(k),(ShiftRows(SubBytes(r)))+(keys[10])}
85: {}
86: {}
88: {}
89: {}
90: {}
91: {}
93: {}
94: {}
95: {}
96: {}
97: {}
99: {}
100: {}
101: {}
102: {}
104: {}
105: {}
106: {}
107: {}
109: {}
110: {}
111: {}
112: {}
113: {}
```

A.2.2 SHA-1

A.2.2.1 Implementação

```
1 typedef byte := unsigned bits[8];
2 typedef word := unsigned bits[32];
```

```
3  typedef wordA := mod[2**32];
4
5  def A : word;
6  def B : word;
7  def C : word;
8  def D : word;
9  def E : word;
10 def W : vector[80] of word;
11
12 def K : vector[4] of word := {
13         word:0x5A827999, word:0x6ED9EBA1, word:0x8F1BBCDC, word:0xCA62C1D6
14 };
15
16 def sha1_compress(Mi : vector[16] of word) : void {
17     def Al, Bl, Cl, Dl, El, T : word;
18
19     W[0..15] := Mi;
20     seq j := 16 to 79 { W[j] := (W[j-3] ^ W[j-8] ^ W[j-14] ^ W[j-16]) <| 1; }
21     Al := A; Bl := B; Cl := C; Dl := D; El := E;
22     seq jj := 0 to 19 {
23         T := word:(wordA:(A <| 5) + wordA:((B&C)|((~B)&D))
24            + wordA:E + wordA:K[0] + wordA:W[jj]);
25         E := D; D := C; C := B <| 30; B := A; A := T;
26     }
27     seq jjj := 20 to 39 {
28         T := word:(wordA:(A <| 5) + wordA:(B^C^D)
29            + wordA:E + wordA:K[1] + wordA:W[jjj]);
30         E := D; D := C; C := B <| 30; B := A; A := T;
31     }
32     seq jjjj := 40 to 59 {
33         T := word:(wordA:(A <| 5) + wordA:((B&C)|(B&D)|(C&D))
34            + wordA:E + wordA:K[2] + wordA:W[jjjj]);
35         E := D; D := C; C := B <| 30; B := A; A := T;
36     }
37     seq jjjjj := 60 to 79 {
38         T := word:(wordA:(A <| 5) + wordA:(B^C^D)
39            + wordA:E + wordA:K[3] + wordA:W[jjjjj]);
40         E := D; D := C; C := B <| 30; B := A; A := T;
41     }
42     A := word:(wordA:A + wordA:Al);
43     B := word:(wordA:B + wordA:Bl);
44     C := word:(wordA:C + wordA:Cl);
45     D := word:(wordA:D + wordA:Dl);
46     E := word:(wordA:E + wordA:El);
47  }
48
49 def sha1_padd(const l : register int { l > 0 },
```

```
50          const nWords : register int { nWords > 0 }, msg : vector[l] of byte)
51          : vector[nWords] of word {
52
53      def c : register int;
54      def M : vector[nWords] of word;
55
56      c := 0;
57
58      seq i := 0 to l / 4 - 1 {
59          M[i] := msg[c+3] @ msg[c+2] @ msg[c+1] @ msg[c];
60          c := c + 4;
61      }
62
63      if (l - c == 0) {
64          M[l / 4] := 0b10000000000000000000000000000000;
65      }
66      else {
67          if (l - c == 1) {
68              M[l / 4] := 0b10000000000000000000000 @ msg[c];
69          }
70          else {
71              if (l - c == 2) {
72                  M[l / 4] := 0b1000000000000000 @ msg[c+1] @ msg[c];
73              }
74              else {
75                  M[l / 4] := 0b10000000 @ msg[c+2] @ msg[c+1] @ msg[c];
76              }
77          }
78      }
79
80      seq ii := (l / 4 + 1) to (nWords - 3) {
81        M[ii] := 0b00000000000000000000000000000000;
82      }
83
84      M[nWords - 2] := word: (int:l * 8 / 0x100000000);
85      M[nWords - 1] := word: (int:l * 8);
86
87      return M;
88  }
89
90
91  def sha1(const l : register int { l > 0 }, msg : vector[l] of byte)
92          : unsigned bits[160] {
93      def M : vector[16] of word;
94      def c, k: register int;
95      def res : unsigned bits[160];
96
```

```
97      def const nBlocks : register int := (l * 8 + 64) / 512 + 1;
98      def blocks : vector[16 * nBlocks] of word;
99      blocks := sha1_padd(l, 16 * nBlocks, msg);
100
101         A := word:0x67452301;
102         B := word:0xEFCDAB89;
103         C := word:0x98BADCFE;
104         D := word:0x10325476;
105         E := word:0xC3D2E1F0;
106
107     seq i := 0 to nBlocks - 1 {
108         M[0..15] := blocks[i * 16 .. i * 16 + 15];
109         sha1_compress(M);
110     }
111
112     res := E @ D @ C @ B @ A;
113
114     return res;
115 }
```

A.2.2.2 Resultado da Análise de Expressões Disponíveis

```
1:  {}
2:  {}
3:  {}
4:  {}
5:  {}
6:  {}
7:  {}
8:  {}
9:  {}
10: {}
11: {}
12: {}
13: {}
14: {}
15: {}
16: {}
17: {}
18: {}
19: {}
20: {}
21: {}
22: {}
23: {}
24: {}
25: {}
```

```
26: {}
27: {}
28: {}
29: {}
30: {}
31: {}
32: {}
33: {}
34: {}
35: {}
36: {}
37: {}
38: {}
39: {}
40: {}
41: {}
42: {}
43: {}
44: {}
45: {}
46: {}
47: {}
48: {}
49: {}
50: {}
51: {}
52: {}
54: {}
55: {}
56: {}
57: {((l)/(4))-(1)}
58: {((l)/(4))-(1)}
59: {((l)/(4))-(1)}
60: {((l)/(4))-(1)}
61: {((l)/(4))-(1)}
62: {((l)/(4))-(1)}
63: {((l)/(4))-(1)}
64: {((l)/(4))-(1)}
65: {((l)/(4))-(1)}
66: {((l)/(4))-(1)}
67: {((l)/(4))+(1),(nWords)-(3),((l)/(4))-(1)}
68: {((l)/(4))+(1),(nWords)-(3),((l)/(4))-(1)}
69: {((l)/(4))+(1),(nWords)-(3),((l)/(4))-(1)}
70: {((l)/(4))+(1),(nWords)-(3),((l)/(4))-(1)}
71: {((l)/(4))+(1),(nWords)-(3),((l)/(4))-(1)}
73: {}
74: {}
```

```
75: {}
76: {(((((l)*(8))+(64))/(512))+(1)}
77: {(((((l)*(8))+(64))/(512))+(1)}
78: {(((((l)*(8))+(64))/(512))+(1)}
79: {(((((l)*(8))+(64))/(512))+(1)}
80: {(((((l)*(8))+(64))/(512))+(1)}
81: {(((((l)*(8))+(64))/(512))+(1)}
82: {(((((l)*(8))+(64))/(512))+(1)}
83: {(((((l)*(8))+(64))/(512))+(1)}
84: {(((((l)*(8))+(64))/(512))+(1),(nBlocks)-(1)}
85: {(((((l)*(8))+(64))/(512))+(1),(nBlocks)-(1)}
86: {(((((l)*(8))+(64))/(512))+(1),(nBlocks)-(1)}
87: {(((((l)*(8))+(64))/(512))+(1),(nBlocks)-(1)}
88: {(((((l)*(8))+(64))/(512))+(1),(nBlocks)-(1)}
91: {}
92: {}
94: {}
95: {}
97: {}
98: {}
100: {}
101: {}
102: {}
103: {}
```