THÈSE DE DOCTORAT DE SORBONNE UNIVERSITÉ

Spécialité
Informatique

École doctorale Informatique, Télécommunications et Électronique (Paris)

Présentée par

Francesco Parolini

Pour obtenir le grade de

DOCTEUR de SORBONNE UNIVERSITÉ

Sujet de la thèse:

Static Analysis for Security Properties of Software by Abstract Interpretation

Thèse soutenue le 26 Juin 2024 devant le jury composé de:

Antoine Miné	Sorbonne Université & CNRS, France	Directeur de thèse
Isabella Mastroeni	Università di Verona, Italie	Rapportrice
Thomas Jensen	Université de Rennes & INRIA, France	Rapporteur
Antoine Genitrini	Sorbonne Université & CNRS, France	Président du jury
Nathalie Sznajder	Sorbonne Université & CNRS, France	Examinatrice
Pierre Ganty	IMDEA Software Institute, Spain	Examinateur
Sébastien Bardin	CEA, List, Université Paris-Saclay, France	Examinateur

Abstract

This thesis aims at developing and implementing formal techniques that can prove the absence of security-related vulnerabilities in software systems. We focus our attention on two notable cases: Regular Expression Denial of Service attacks (ReDoS), and exploitable runtime errors. For each case, we first study the theoretical framework to precisely characterize the vulnerability that we are considering. Then, we develop *sound, automatic* analyses, which can prove the absence of the vulnerabilities without relying on user interaction or annotations. We pair our theoretical results with practical implementations, which we consistently test on real-world examples.

Modern programming languages often provide functions to manipulate regular expressions in standard libraries. If they offer support for advanced features, the matching algorithm has an exponential worst-case time complexity: for some so-called vulnerable regular expressions, an attacker can craft ad hoc strings to force the matcher to exhibit the exponential behaviour and perform a ReDoS attack. In the first part of this thesis, we put forward a novel *tree semantics* for the regular expression matching procedure that precisely characterizes the behaviour of real-world matching engines. By leveraging such a characterization, we formally define ReDoS vulnerabilities in terms of the size of the matching trees. Then, we propose a sound analysis which extracts an overapproximation of the set of words that can make the matching engine exhibit the exponential behaviour. We implemented our analysis in a tool called RAT, and by comparing it to seven other detectors on a large set of 74,669 regular expressions, we found that RAT is the only detector that does not raise false negatives. Furthermore, RAT is faster, often by orders of magnitude, than most other tools.

Runtime errors that can be triggered by an attacker are sensibly more dangerous than others, as they not only result in program failure, but can also be exploited and lead to security breaches. In the second part of this thesis, we focus our attention on developing a technique able to rule out the existence of runtime errors that can be triggered by an attacker. First, we introduce a novel property called *safety-nonexploitability*, which precisely characterizes the set of programs whose correctness cannot be altered by an external user. Then, we give an alternative characterization of safety-nonexploitability in terms of tainted (i.e., user-controlled) variables. By relying on this characterization, we propose an analysis by abstract interpretation which combines a semantic taint analysis with a numeric analysis. The numeric invariants inferred by the numeric domain enhance the precision of the taint analysis. To assess the usefulness of our technique, we implemented our analysis for a large subset of C. We compared the regular analyzer with the modified nonexploitability version on a large set of 77 real-world programs taken from the Coreutils package, to which we added 13,261 test cases taken from the Juliet test suite. In our experiments, we found that our framework can consistently prove that more than 70% of the alarms raised by the regular analyzer are not exploitable.

Résumé

Cette thèse vise à développer et à mettre en œuvre des techniques formelles capable de prouver l'absence de vulnérabilités liées à la sécurité dans les systèmes logiciels. Nous concentrons notre attention sur deux cas notables: les attaques Déni de Service liées aux Expressions Régulières (ReDoS), et les erreurs à l'exécution qui peuvent être déclenchées par un attaquant. Pour chaque cas, nous étudions d'abord le cadre théorique pour caractériser précisément la vulnérabilité que nous considérons. Ensuite, nous développons des analyses sûres et automatiques, qui peuvent prouver l'absence de vulnérabilités sans requérir d'interaction ou d'annotations de l'utilisateur.

Les langages de programmation modernes fournissent souvent des fonctions permettant de manipuler les expressions régulières. Lorsqu'elles offrent une prise en charge de fonctionnalités avancées, l'algorithme de correspondance a une complexité en temps dans le pire des cas exponentielle: pour certaines expressions régulières dites vulnérables, un attaquant peut alors créer des chaînes ad hoc pour forcer le moteur de recherche d'expressions régulières à présenter un comportement exponentiel et ainsi réaliser une attaque ReDoS. Dans la première partie de cette thèse, nous proposons une nouvelle sémantique pour la procédure de correspondance des expressions régulières qui caractérise précisément le comportement des moteurs de recherche d'expressions régulières réels. En exploitant une telle caractérisation, nous définissons formellement les vulnérabilités ReDoS en termes de taille des arbres de correspondance. Ensuite, nous proposons une analyse sûre qui extrait une surapproximation de l'ensemble des mots pouvant entraîner un comportement exponentiel du moteur de recherche d'expressions régulières. Nous avons implémenté notre analyse dans un outil appelé RAT, et en le comparant à sept autres détecteurs sur un large ensemble de 74,669 expressions régulières, nous avons constaté que RAT est le seul détecteur à ne pas générer de faux négatifs. De plus, RAT est plus rapide, souvent de plusieurs ordres de grandeur, que la plupart des autres outils.

Les erreurs à l'exécution pouvant être déclenchées par un attaquant sont sensiblement plus dangereuses que d'autres, car elles entraînent non seulement un échec du programme, mais peuvent également être exploitées et conduire à des violations de sécurité. Dans la deuxième partie de cette thèse, nous concentrons notre attention sur le développement d'une technique capable de prouver l'absence d'erreurs à l'exécution pouvant être déclenchées par un attaquant. Nous introduisons une nouvelle propriété appelée *non-exploitabilité,* qui caractérise précisément l'ensemble des programmes dont la correction ne peut pas être altérée par un utilisateur externe. Ensuite, nous donnons une caractérisation alternative de la non-exploitabilité en termes de variables *teintées* (c'est-à-dire contrôlées par l'utilisateur). En nous appuyant sur cette caractérisation, nous proposons une analyse par interprétation abstraite qui combine une analyse sémantique de teinte avec une analyse numérique. Les invariants numériques déduits par le domaine numérique améliorent la précision de l'analyse de teinte. Pour démontrer l'utilité de notre technique, nous avons implémenté notre analyse pour un large sous-ensemble de C. Nous avons comparé l'analyseur original avec la version modifiée par la non-exploitabilité sur un grand ensemble de 77 programmes réels extraits du package Coreutils, auxquels nous avons ajouté 13,261 cas de tests issus de la suite de tests Juliet. Dans nos expériences, nous avons constaté que notre analyse peut systématiquement prouver que plus de 70% des alarmes soulevées par l'analyseur original ne sont pas exploitables.

Publications

First-author publications

Conference papers

Sound Static Analysis of Regular Expressions for Vulnerabilities to Denial of Service Attacks
Francesco Parolini and Antoine Miné
in Theoretical Aspects of Software Engineering (TASE) 2022

Sound Abstract Nonexploitability Analysis
Francesco Parolini and Antoine Miné
in Verification, Model Checking, and Abstract Interpretation (VMCAI) 2024

Journal papers

Sound Static Analysis of Regular Expressions for Vulnerabilities to Denial of Service Attacks (Extended Version)
Francesco Parolini and Antoine Miné
in Science of Computer Programming 2023

Other publications

Inclusion Testing of Büchi Automata Based on Well-Quasiorders
Kyveli Doveri, Pierre Ganty, Francesco Parolini, Francesco Ranzato
in International Conference on Concurrency Theory (CONCUR) 2021

Mopsa-C: Improved Verification for C Programs, Simple Validation of Correctness Witnesses (Competition Contribution)

Raphaël Monat, Marco Milanese, Francesco Parolini, Jérôme Boillot, Abdelraouf Ouadjaout, Antoine Miné

in Tools and Algorithms for the Construction and Analysis of Systems (TACAS) 2024

Acknowledgments, Remerciements & Ringraziamenti

I would like to start by thanking Isabella Mastroeni and Thomas Jensen for reviewing this manuscript. I would also like to thank Antoine Genitrini, Nathalie Sznajder, Pierre Ganty, and Sébastien Bardin for agreeing to be part of the PhD jury. Thanks to everyone for the comments and the interesting discussions about my work.

Merci Antoine de m'avoir accepté comme doctorant et de m'avoir donné d'innombrables conseils, orientations et enseignements. Tu as toujours trouvé le temps pour de longues discussions, et tes notes et tes relectures ont toujours été extrêmement détaillées et précises. Sans ton engagement, je n'aurais pas pu terminer ce manuscrit. Merci d'avoir été un directeur de thèse si exceptionnel.

Grazie a Francesco Ranzato per avermi fatto appassionare all'analisi statica. Il suo corso di Verifica del Software ha sicuramente cambiato il corso della mia vita (non starei scrivendo questi ringraziamenti altrimenti!), ed è stato un piacere lavorare con lei durante la mia tesi magistrale. Non è facile trovare professori che facciano appassionare gli studenti a quello che insegnano, e io sono stato fortunato.

Grazie a Gregorio Piccoli per avermi mostrato il mondo degli interpreti e dei linguaggi funzionali. Lo stage che mi ha proposto mi ha fatto innamorare dei linguaggi di programmazione, e questo amore continua tutt'oggi.

Thanks to Abdelraouf, Matthieu, Raphaël, David, Guillaume, Marco, and Milla for being part of the Mopsa team. It has been a pleasure working with such a talented group of individuals. Thanks to Kyveli for leading the research project in the first paper I ever published.

Thanks to my (ex) colleagues and friends at Amazon: Pauline, Vlad, Stefan, Horia, Daniel T., Sandro, Evan, Franco, Djordje, Claudia, Norine, Philipp, and Sarek. Thanks to Ilina and Daniel for being such great bosses and functional programming enthusiasts.

Merci à tous les doctorants de la salle 303: Jules, Martin, Mathieu, Mamy, Mohamed, Keanu et Sébastien. Thanks also to the PhD students from our sister lab at ENS: Jérôme,

Charles, Josselin, Valentin, Ignacio, Albin, and Patricio. Grazie agli altri ricercatori e amici, Alessio, Adam, Marco C., Alessandro e Gabriele. Together, we spent many fun Parisian nights, and you all really made my stay in France special. I'm sure we will meet again at Dimitry's.

Thanks to the friends that I met along the away abroad, in particular Maria and Alvaro. You made me feel at home, even thousands of kilometers away. Grazie a tutti i miei amici incontrati all'università: Alessandro, Paolo, Andrea, Alessio, Simone, Ciprian, e Linpeng. Grazie a Denis per avermi convinto a trasferirmi a Parigi e avermi accompagnato in questa avventura. Grazie a tutte le altre persone speciali che mi hanno accompagnato in questo viaggio: Marta, Matteo, Silvia, Dario, Gualtio, Alessio, Valerio, e Federica. Ci siete stati fin dall'inizio, e so che ci sarete anche in futuro.

Thanks to all the people that I forgot to thank.

Grazie a tutta la mia famiglia, in particolare a mia zia Paola, mia cugina Anna, nonna Domenica, zio Giorgio, e ai miei genitori Alba e Gian. Senza di voi, non avrei potuto fare nulla.

Ai miei genitori

Contents

Part I

Background

Chapter 1

Introduction

Software systems are nowadays ubiquitous. Programs control safety-critical systems such as airplanes [1], emergency systems of nuclear power plants [2], and car engines [3]. Furthermore, businesses use software to gather possibly sensitive data from their users, and they are legally obliged to guarantee the secrecy of such information [4]. The growing complexity of software systems progressively increases the likelihood for programmers to introduce software errors. Programs with bugs can lead to huge economic losses [5, 6, 7], and, in some cases, they can even result in loss of human lives [8, 9].

Recent advancements in artificial intelligence saw the rise of large language models (LLMs). These systems are capable of writing software starting from a specification written in natural language. Conversational agents such as OpenAI's ChatGPT [10] and Google's Gemini [11] have been attracting a lot of attention from the public recently, and their use is becoming increasingly integrated into people's everyday lives. Some LLMs are specifically designed to write computer programs: Github Copilot [12] is an AI-based coding assistant that integrates into IDEs and assists the programmer during the development process. Github Copilot has currently over 1 million paid subscribers in over 37,000 organizations [13]. However, there is growing evidence that AI-generated code introduces at least as many bugs as programmers [14, 15, 16, 17]. In this scenario, it is paramount to employ techniques to find software errors early in the development cycle.

1.1. Approaches to reliable software

In order to find bugs before deploying an application, programmers use software engineering techniques such as *testing*. The correct behaviour of a program is tested for a finite–and often relatively small–number of inputs, and assertions in the test suite check the functional correctness of a piece of code. As every test case has to be manually written by the programmer, this technique is time-consuming. Furthermore, exhaustively testing all the possible program inputs is not feasible, so that the correctness is checked only for a small, hopefully representative, portion of the input space. As Dijkstra famously remarked "testing can be quite effective for showing the presence of bugs, but is hopelessly inadequate for showing their absence" [18].

During recent years, language-based techniques to enhance software reliability have begun to gain some traction. *Strong type systems* are nowadays common, and widely-used modern programming languages such as Scala [19], Rust [20], and Swift [21] adopt a strict typing discipline. Relevant properties such as *null-safety* (i.e., the absence of null-pointer dereferences) can be guaranteed at compile-time by strong type systems, and their use is increasingly popular. Another significant example of property that can be guaranteed at compile-time is *memory safety* in non-garbage collected languages, which has been recently popularized by the Rust programming language. While these strong type systems prevent some failures to occur, they cannot completely rule out the existence of all errors, which is beyond the scope and capabilities of those systems.

Although testing and a strong typing discipline can prevent some errors to occur, they cannot ultimately eliminate all bugs. On the other hand, *formal methods* provide strong mathematical guarantees about the correctness of software. By relying on a precise mathematical model of the programs' behaviour, formal methods can reason about the semantics of software systems, and this makes it possible to prove properties of programs. However, Rice's undecidability theorem [22] states that all non-trivial program properties are undecidable, and poses a major challenge to program verification. To elude Rice's theorem, formal method techniques sacrifice either *completeness* (all true facts are provable), *soundness* (the conclusions about programs are always correct under suitable explicitly stated hypotheses), or *automation* (proofs are carried out by a computer).

Deductive methods produce proofs of correctness, but ultimately require user interaction. This approach makes it possible to prove strong properties of programs, such as functional correctness with respect to a specification. Even if proof assistants help the user with hints and strategies to carry out a proof, this process cannot be ultimately fully automatized.

Symbolic execution techniques perform an abstract execution of programs by assuming symbolic variables for the unknown values, and propagate them during the analysis. The collected constraints are precise, and can be solved to determine if an arbitrary assertion is violated (e.g., absence of runtime errors). Since the number of feasible execution paths grows exponentially with the size of programs, symbolic execution techniques have sometimes to trade soundness for performance [23].

Model checking restricts the verification problem to decidable fragments of languages [24] and produces correctness proofs automatically. Clarke et al. [25] apply *bounded model checking* to prove the correctness of ANSI-C programs. Their approach unwinds loops and function calls up to a threshold, which implies that behaviours beyond such a threshold are not considered.

Abstract interpretation [26] is a formal technique to automatically prove the correctness of software systems. *Abstract interpreters*, i.e., analyzers that rely on abstract interpretation theory, run an abstract execution on programs and collect an overapproximation of the reachable states. In a single run, they consider all concrete executions, to which they necessarily add some spurious, hopefully irrelevant, ones. While analyses derived from the abstract interpretation framework are *not complete* (i.e., they can present *false positives*), once the analyzer classifies a program as safe, then there is a strong mathematical guarantee about the correctness of the system (i.e., *false negatives* are forbidden). Abstract interpreters target specific kinds of formally defined undesirable behaviours, and can only ensure the absence of these errors (e.g., index out-of-bounds, null pointer dereferences). During the years, many static analyzers based on abstract interpretation theory have been developed. For example, the ASTRÉE [27] analyzer is a commercial static analysis tool specifically designed to verify the correctness of large embedded safety-critical software written in C. The analyzer was able to prove the absence of runtime errors in the flight control codes of the Airbus fly-by-wire systems. More recently, the FRAMA-C abstract interpretation plugin has been used to analyze

software that manages nuclear power plants [28].

1.2. The challenges of cybersecurity

A particularly interesting class of software flaws is *security-related vulnerabilities*. In non-safety critical applications, security vulnerabilities are often considered more dangerous than runtime errors, despite a large class of security breaches being caused by such errors [29]. For instance, banking applications employ an extensive security audit process by cybersecurity experts to ensure that sensible financial data is not leaked to unauthorized users [30, 31, 32]. Numerous companies sell cybersecurity-related services, universities offer degrees in security, and virtually every country's government has established agencies specifically dedicated to addressing cyber threats.

In order to find security vulnerabilities early in the development process, a software system is tested with well-established techniques such as *penetration testing* [33, 34] and *code auditing* by security experts. Nevertheless, similarly to runtime errors, these techniques cannot mathematically guarantee that an application is secure, making formal methods the only viable option for formally ensuring security.

One of the main challenges that security poses to formal methods is mathematically defining when an application is *secure*. While programs that present classic runtime failures such as null pointer dereferences and index out-of-bounds are clearly wrong, it is more difficult to semantically classify programs as *secure* or not. The reason is that security ranges over a wide spectrum of high-level properties, including *confidentiality* (absence of secret information leakage), *integrity* (data remains unaltered during storage, transmission, and processing), and *authentication* (the entities that interact with the system can be reliably identified).

1.3. Contributions and outline

In this thesis, we put forward techniques based on formal reasoning to detect security-related vulnerabilities. The analyses we propose are *semantic*, namely grounded in the mathematical description of software systems' behaviour. By

reasoning on the concrete semantics, the techniques we propose are able to *prove the absence of security-related vulnerabilities*. Part I gives an introduction to this work, and in particular in Chapter 2 we provide the mathematical background used in the rest of this thesis. In the remainder of this section, we outline the organization of the rest of this manuscript.

1.3.1. Verification of security properties for regular expressions

In Part II of this manuscript, we focus our attention on proving the absence of Regular Expression Denial of Service attacks (ReDoS), a particular type of *algorithmic complexity attacks* [35]. Matching engines in languages such as Python, JavaScript, and Java employ algorithms with exponential worst-case time complexity in the length of the string. An attacker can craft ad-hoc strings to exploit such vulnerabilities and make the matching engine exhibit the exponential behaviour. ReDoS attacks are a vastly underestimated class of Denial of Service attacks, affecting over 10% of Node.js-based web services [36]. Many well-known platforms such as Stack Overflow [37], Cloudflare [38], and iCloud [39] have reported exponential matching in their systems.

In Chapter 3, we give an introduction to regular expressions and automata. Then, Chapter 4 introduces a formal semantics for regular expression matching, which is then used to put forward a sound analysis for ReDoS vulnerabilities. If our algorithm classifies a regular expression as safe, then there is a strong mathematical guarantee about the fact that the exponential behaviour cannot be triggered. In order to assess the usefulness of our technique, we implemented it in a tool called RAT [40]. In Chapter 5 we compare our analyzer to seven other ReDoS detectors on a large dataset of 74,669 regular expressions. In our experiments, we observed that our implementation is the only ReDoS analyzer that does not present false negatives. Furthermore, in the great majority of the test cases, RAT is faster–often by orders of magnitude–than most other tools.

1.3.2. Verification of security properties for programs

In Part III of this thesis, we shift our focus to program analysis. A particularly interesting class of runtime failures is those that can be triggered by an external user. These errors are more dangerous than those that cannot, as they could be exploited

by an attacker and lead to security breaches such as Denial-of-Service attacks or remote code execution. There are many well-known exploitable runtime errors that led to sophisticated exploits. Among them, we find Code Red [41], the Morris Worm [42], SQL Slammer [43], and Heartbleed [44]. Numerous companies identified exploitable runtime errors in their systems, including Meta [45], Apple [46], and Google [47]. Microsoft recently published a report showing that consistently over 20 years, around 70% of the security breaches that have been reported in their systems are due to exploitable memory corruption [29]. In our work, we are interested in proving the *nonexploitability* of a system, namely verifying the absence of runtime errors that can be triggered by an attacker.

In Chapter 6 we give a lightweight introduction to program analysis by abstract interpretation. Then, in Chapter 7 we put forward our framework for the nonexploitability analysis. First, we introduce the novel property of *safety-nonexploitability*, which we show can be characterized in terms of semantically tainted (i.e., user-controlled) variables. We leverage this characterization to design a sound analysis by abstract interpretation capable of ruling out the presence of exploitable runtime error. The analysis combines a semantic taint analysis–namely an analysis that tracks the set of user-controlled variables–with a traditional value analysis. As a result, the precision of the taint analysis is enhanced by the program invariants inferred by the value domain. The analysis has the capability to label each warning as security-critical or not, prioritizing the alarms by possible impact and therefore enhancing the usefulness of the analyzer. To evaluate the effectiveness of our framework, in Chapter 8 we implement it for a large subset of C by relying on MOPSA [48], a platform to build static analyses based on abstract interpretation. We compare the regular analysis and our modified version on 77 real-world programs taken from the Coreutils package. To them, we add 13,261 test cases from the Juliet benchmarks [49]. We found that our analysis is able to consistently *prove* that more than 70% of the alarms generated by the regular analyzer are not exploitable.

1.3.3. Contributions

In this work, we put forward techniques based on formal reasoning to rule out the existence of security vulnerabilities in the context of both regular expressions and program runtime errors. Our analyses can possibly raise false alarms, but once

they classify a system as secure, then there is a strong mathematical guarantee about the fact that there are no security breaches concerning the properties we consider. We pair our theoretical frameworks with practical implementations, which we consistently test on real-world examples. In this thesis, we claim the following contributions:

- In Part II of this manuscript:
  - We introduce a novel *tree semantics* to describe the behaviour of regular expression matching engines, and we leverage it to formally define ReDoS vulnerabilities.
  - We put forward a sound analysis that extracts an overapproximation of the language of words that can cause an exponential ReDoS attack for a regular expression.
  - We implement the analysis in a tool called RAT. We also compare the performance and the precision of RAT to seven other detectors. In our evaluation, we find that RAT is on average one to two orders of magnitude faster than most other approaches, while being strictly more expressive than the others. More interestingly, RAT is the only detector that does not report false negatives.
- In Part III of this manuscript:
  - We introduce a novel property, *safety-nonexploitability*, and we give its semantic characterization as a hyperproperty [50].
  - We put forward an alternative characterization of safety-nonexploitability in terms of *semantically tainted* (i.e., user-controlled) variables.
  - We introduce a new practical, modular analysis by abstract interpretation that combines a traditional value analysis with a taint analysis to prove safety-nonexploitabilty.
  - We implement our analysis and evaluate it on a large set of real-world C programs. In our experiments, we found that our framework is able to consistently prove that more than 70% of the alarms previously raised by the regular analyzer are not exploitable.

The results described in Part II have been published in TASE 2022 [51], and subsequently extended in Science of Computer Programming (2023) [52]. The source code of the RAT analyzer is available on Github [40]. Part III of this thesis is based on a paper that appeared in VMCAI 2024 [53]. An artifact is available on Zenodo to replicate our experimental results [54]. The source code of the analyzer is available on Gitlab [55].

Chapter 2

Mathematical Background

In this chapter, we study the mathematical background used in the rest of this thesis. We start with basic set theory in Section 2.1, and we then proceed to order theory in Section 2.2. Then, we study the theory of fixpoints in Section 2.3, which is widely used in static analysis and verification.

2.1. Basics

Sets. Let X, Y and Z be three sets. The *Cartesian product* of X and Y is the set $X \times Y \triangleq \{ (x, y) \mid x \in X \wedge y \in Y \}$. The set $X \times X$ is sometimes denoted as X^2. The Cartesian product can be generalized to n-ary tuples, $n \in \mathbb{N}$, as $(x_1, x_2, \ldots, x_n) \in X_1 \times X_2 \times \cdots \times X_n$, and X^n if $X_1 = X_2 = \cdots = X_n$. The *infinite Cartesian product* of a set X is $X \times X \times \cdots \triangleq X^\infty$. Sequences of arbitrary, but finite, length are elements in $X^* \triangleq \bigcup_{n \geq 0} X^n$, where $X^0 \triangleq \emptyset$. Finite sequences of length at least one are elements in $X^+ \triangleq \bigcup_{n \geq 1} X^n$. If $x = x_1, \ldots, x_n \in X_1 \times \cdots \times X_n$ is a tuple of length n, we define $\pi_i : X_1 \times \cdots \times X_n \to X_i$ as $\pi_i(x) \triangleq x_i$ for $1 \leq i \leq n$. We denote by $|X|$ the *cardinality* (i.e., the number of elements) of X, and by $\wp(X)$ its *powerset* (i.e., the set of subsets of X, that is $\{ X' \mid X' \subseteq X \}$). A *partition* P of X is a set of non-empty subsets of X, called *blocks*, that are pairwise disjoint and the union of which gives X.

Relations. A *relation* R over two sets X and Y is a subset of $X \times Y$, namely $R \in \wp(X \times Y)$. The *composition* of two relations $R_1 \in \wp(X \times Y)$, $R_2 \in \wp(Y \times Z)$ is defined as $R_1 \circ R_2 \triangleq \{ (x, z) \mid \exists (x, y) \in R_1 \wedge \exists (y, z) \in R_2 \}$. A relation $R \in \wp(X \times Y)$ is *total* if

$\forall x \in X : \exists y \in Y : (x, y) \in R$. Let R be a relation on X^2.

- R is *reflexive* if $\forall x \in X : (x, x) \in R$.
- R is *transitive* if $\forall x_1, x_2, x_3 \in X : (x_1, x_2) \in R \land (x_2, x_3) \in R \implies (x_1, x_3) \in R$.
- R is *symmetric* if $\forall x_1, x_2 \in X : (x_1, x_2) \in R \iff (x_2, x_1) \in R$.
- R is *antisymmetric* if $\forall x_1, x_2 \in X : (x_1, x_2) \in R \land (x_2, x_1) \in R \implies x_1 = x_2$.

If R is reflexive, transitive, and symmetric we say that R is an *equivalence*. If R is reflexive, transitive, and antisymmetric we say that R is a *partial order*. If R is reflexive and transitive we say that R is a *quasiorder* (or *preorder*).

Functions. A *function* is a relation $R \in \wp(X \times Y)$ where any element of X is in relation with *at most* one element of Y, that is $\forall x \in X : \forall y_1, y_2 \in Y : (x, y_1) \in R \land (x, y_2) \in R \implies y_1 = y_2$. If $f \in \wp(X \times Y)$ is a function from X to Y we write $f : X \to Y$. A function $f : X \to Y$ is *partial* if $\exists x \in X : \nexists y \in Y : f(x) = y$. A function $f : X \to X$ where the domain and codomain coincide is sometimes called an *operator*.

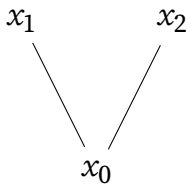
The composition of two functions $f : X \to Y$ and $g : Y \to Z$ is defined as $(g \circ f)(x) \triangleq g(f(x))$. We define $id : X \to X$ as the *identity function*, namely $id(x) \triangleq x$. Let $f : X \to X$ be a function. For all $n \in \mathbb{N}$ we inductively define:

$$f^n \triangleq \begin{cases} id & \text{if } n = 0 \\ f \circ f^{n-1} & \text{if } n > 0 \end{cases}$$

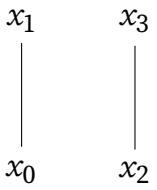
We define the following classes of functions:

$$\mathcal{O}(g) \triangleq \{ f \mid \exists c_1 : \exists n_0 : \forall n \geq n_0 : f(n) \leq c_1 \cdot g(n) \}$$
$$\Omega(g) \triangleq \{ f \mid \exists c_1 : \exists n_0 : \forall n \geq n_0 : f(n) \geq c_1 \cdot g(n) \}$$
$$\Theta(g) \triangleq \{ f \mid \exists c_1 : \exists c_2 : \exists n_0 : \forall n \geq n_0 : c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \}$$

If $f \in \mathcal{O}(g)$, we write $f = \mathcal{O}(g)$.



A. Poset with three elements



B. Poset with four elements

FIGURE 2.1. Examples of posets

2.2. Order theory

Posets. Posets are defined as sets equipped with a *partial order*.

Definition 2.1 (Poset)
A *partially ordered set* (poset) $(X, \sqsubseteq)$ is a set X equipped with a partial order relation $\sqsubseteq \in \wp(X \times X)$.

Example 2.1 (Poset)
Let X be a set. Then, $(\wp(X), \subseteq)$ is a poset ordered by set inclusion. Observe that there is no need for X to be a poset itself.

Let $(X, \sqsubseteq)$ be a poset. Two elements $x_1, x_2 \in X$ are *comparable* if either $x_1 \sqsubseteq x_2$ or $x_2 \sqsubseteq x_1$ holds, and they are *incomparable* otherwise. *Hasse diagrams* are used to graphically represent posets. In such diagrams, each element of X corresponds to a vertex in the plane, and draws a segment that goes upward from one vertex x_1 to another vertex x_2 whenever $x_1 \sqsubseteq x_2$.

Example 2.2 (Hasse diagram)
Figure 2.1A-2.1B represent examples of posets. In particular, Figure 2.1A represents the poset $(\{x_0, x_1, x_2\}, \{(x_0, x_1), (x_0, x_2)\})$, and Figure 2.1B represents the poset $(\{x_0, x_1, x_2, x_3\}, \{(x_0, x_1), (x_2, x_3)\})$.

Lower and upper bounds. Let $(X, \sqsubseteq)$ be a poset and $X' \subseteq X$ be a subset of X. The subset X' has:

- An *upper bound* u iff $u \in X$ and $\forall x \in X' : x \sqsubseteq u$

- A *least upper bound* (lub) $\sqcup X'$ iff $\sqcup X'$ is an upper bound of X' smaller than other upper bounds of X'. We denote $\sqcup\{x_1, x_2\}$ as $x_1 \sqcup x_2$.
- A *top element* $\top$ iff $\top = \sqcup X \in X$
- A *lower bound* l iff $l \in X$ and $\forall x \in X' : l \sqsubseteq x$
- A *greatest lower bound* (glb) $\sqcap X'$ iff $\sqcap X'$ is a lower bound of X' greater than other lower bounds of X'. We denote $\sqcap\{x_1, x_2\}$ as $x_1 \sqcap x_2$.
- A *bottom element* $\bot$ iff $\bot = \sqcap X \in X$

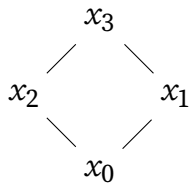
Lattices. Lattices are posets that require every nonempty finite subset to have a lub and a glb.

Definition 2.2 (Lattice)
A *lattice* $(X, \sqsubseteq, \sqcup, \sqcap)$ is a poset where $\forall x_1, x_2 \in X$ the lub $x_1 \sqcup x_2$ and the glb $x_1 \sqcap x_2$ exist.

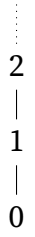
Complete lattices are lattices that additionally require every (possibly infinite) subset to have a lub and a glb.

Definition 2.3 (Complete lattice)
A *complete lattice* $(X, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$ is a poset where $\forall X' \subseteq X$ the lub $\sqcup X'$ and the glb $\sqcap X'$ exist. Complete lattices have a *top* element $\top \triangleq \sqcup X$ and a *bottom* element $\bot \triangleq \sqcap X$.

Example 2.3 (Lattices)
The posets in Figure 2.1A and Figure 2.1B are not lattices, as they do not have a lub for every possible finite subset. Figure 2.2A represents a complete lattice with a finite number of elements. The set $(\mathbb{N}, \leq)$ (represented in Figure 2.2B) is a lattice, but not a complete lattice: while every *finite* subset has a lub, $\sqcup\mathbb{N}$ does not exist. If we add ∞ to $\mathbb{N}$, we observe that $(\mathbb{N} \cup \{\infty\}, \leq)$ (represented in Figure 2.2C) is indeed a complete lattice.



A. Complete lattice with a finite number of elements



B. Lattice with an infinite number of elements

C. Complete lattice with an infinite number of elements

FIGURE 2.2. Examples of lattices

CPOs. A *chain* C of a poset $(X, \sqsubseteq)$ is a subset of X such that any two elements are comparable: $C \subseteq X$ is a *chain* $\overset{\triangle}{\iff} \forall x_1, x_2 \in C : x_1 \sqsubseteq x_2 \vee x_2 \sqsubseteq x_1$. An *ascending chain* is a sequence $x_0, x_1, \ldots$ such that $x_0 \sqsubseteq x_1 \sqsubseteq \cdots \sqsubseteq x_{n-1} \sqsubseteq x_n \sqsubseteq \cdots$.

Example 2.4 (Infinite ascending chain)
Let $\text{fib}(n)$ be the n-th Fibonacci number defined as follows.

$$\text{fib}(n) \triangleq \begin{cases} n & \text{if } n = 0 \text{ or } n = 1 \\ \text{fib}(n-1) + \text{fib}(n-2) & \text{otherwise} \end{cases}$$

The sequence $(\text{fib}(i), i \in \mathbb{N})$ is an infinite ascending chain over $(\mathbb{N}, \leq)$:

$$0 \leq 1 \leq 1 \leq 2 \leq 3 \leq 5 \leq 8 \leq \cdots$$

Definition 2.4 (CPO)
A *complete partial order* (CPO) is a poset such that every chain has a lub.

Observe that the empty sequence $\emptyset$ is a chain, so that our definition of CPO requires $\sqcup\emptyset$ to exist. For this reason, we state that $\sqcup\emptyset = \bot$. As a result, all our CPOs have a bottom element $\bot$.

Monotonicity and continuity. Let $f : X_1 \to X_2$ be a function between two posets $(X_1, \sqsubseteq_1)$ and $(X_2, \sqsubseteq_2)$. We say that f is *monotonic* iff $\forall x_1, x_2 \in X_1 : x_1 \sqsubseteq_1 x_2 \implies f(x_1) \sqsubseteq_2 f(x_2)$. *Continuity* generalizes monotonicity: a function $f : X_1 \to X_2$ between two CPOs $(X_1, \sqsubseteq_1, \sqcup_1, \bot_1)$ and $(X_2, \sqsubseteq_2, \sqcup_2, \bot_2)$ is *continuous* iff for every

chain $C \subseteq X_1$, the set $\{ f(x) \mid x \in C \}$ is a chain and the limits coincide $f(\sqcup_1 C) = \sqcup_2 \{ f(x) \mid x \in C \}$.

2.3. Fixpoints

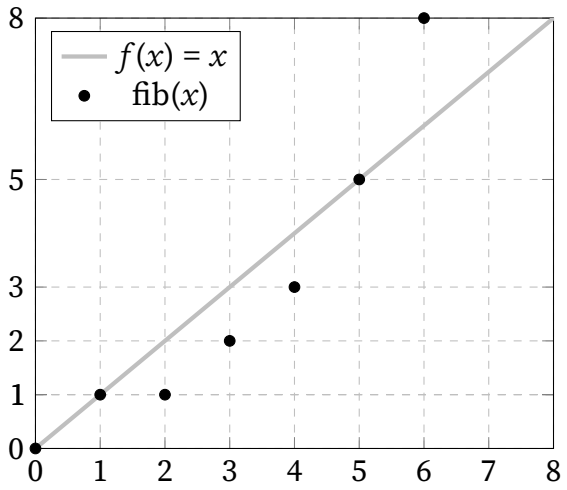
Fixpoints (namely elements such that $f(x) = x$ for an operator $f : X \to X$) are fundamental to formally reason about the behaviour of programming languages. In fact, the *formal semantics* (that is, a precise mathematical description of the behaviour) of programming languages can be expressed as a *least fixpoint*. In this section, we study fixpoints, giving sufficient conditions for their existence.

Definition 2.5 (Fixpoints)
Let $(X, \sqsubseteq)$ be a poset and $f : X \to X$ be an operator on X.

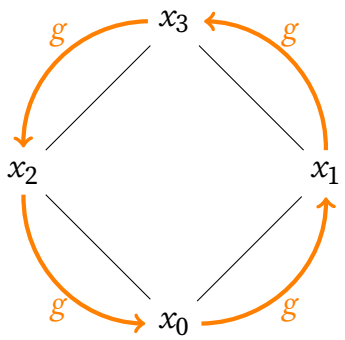
- x is a *fixpoint* of f iff $f(x) = x$
- x is a *prefixpoint* of f iff $x \sqsubseteq f(x)$
- x is a *postfixpoint* of f iff $f(x) \sqsubseteq x$
- $\mathrm{fps}(f) \triangleq \{ x \mid f(x) = x \}$ is the set of fixpoints of f
- $\mathrm{lfp}_{x_0} f \triangleq \min\{ x \in \mathrm{fps}(f) \mid x_0 \sqsubseteq x \}$, if it exists, is the *least fixpoint of f greater than x_0*
- $\mathrm{lfp}\, f \triangleq \mathrm{lfp}_{\bot} f$, if it exists, is the *least fixpoint of f*
- $\mathrm{gfp}_{x_0} f \triangleq \max\{ x \in \mathrm{fps}(f) \mid x \sqsubseteq x_0 \}$, if it exists, is the *greatest fixpoint of f smaller than x_0*
- $\mathrm{gfp}\, f \triangleq \mathrm{gfp}_{\top} f$, if it exists, is the *greatest fixpoint of f*

Observe that, in the general case, there is no guarantee that an operator f has any fixpoint at all.

Example 2.5 (Fixpoints in Fibonacci's sequence)
Figure 2.3A represents the first 6 elements of the Fibonacci sequence as



A. Plot representing the first 6 elements of the Fibonacci sequence



B. Operator with no fixpoints

FIGURE 2.3. Examples of operators

defined in Example 2.4. The set of fixpoints for fib is $\text{fps}(\text{fib}) = \{0, 1, 5\}$.

Example 2.6 (Operator with no fixpoints)
Figure 2.3B represents an operator g over the poset represented in Figure 2.2A, where $g \triangleq \{(x_0, x_1), (x_1, x_3), (x_3, x_2), (x_2, x_0)\}$. The operator g does not have any fixpoint.

We now present two theorems that guarantee, under some assumptions, that fixpoints do exist. The first, *Tarski's fixpoint theorem* [56], relates the monotonicity of an operator in a complete lattice with the existence of a least fixpoint.

Theorem 2.1 (Tarski's fixpoint theorem [56])
Let $f : X \to X$ be a monotonic operator over a complete lattice $(X, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$. Then, the set of fixpoints $\text{fps}(f)$ is a non-empty lattice. In particular, $\text{lfp}\ f$ exists, and it corresponds to the glb of the postfixpoints of f.

$$\text{lfp}\ f = \sqcap\{x \in X \mid f(x) \sqsubseteq x\}$$

Thm. 2.1 is significant because it gives a precise characterization of the least

fixpoint of a monotonic operator as the glb of its postfixpoints.

Example 2.7 (Tarski's least fixpoint)
Consider $\text{fib} : \mathbb{N} \cup \{\infty\} \to \mathbb{N} \cup \{\infty\}$ defined as follows:

$$\text{fib}(n) \triangleq \begin{cases} n & \text{if } n = 0 \text{ or } n = 1 \\ \text{fib}(n-1) + \text{fib}(n-2) & \text{if } n \in \mathbb{N} \setminus \{0, 1\} \\ \infty & \text{if } n = \infty \end{cases}$$

Observe that $(\mathbb{N} \cup \{\infty\}, \leq, \max, \min, 0, \infty)$ is a complete lattice, and fib is a monotonic operator over it. Then, we can apply Tarski's fixpoint theorem and, by observing that the set of postfixpoints for fib is $\{0, 1, 2, 3, 4, 5\}$, obtain the following:

$$\text{lfp fib} = \min\{0, 1, 2, 3, 4, 5\} = 0$$

Another theorem that not only guarantees the existence of the least fixpoint of a function, but also expresses it as the limit of an iteration, is *Kleene's fixpoint theorem* [57].

Theorem 2.2 (Kleene's Fixpoint Theorem [57])
Let $f : X \to X$ be a continuous operator over a CPO $(X, \sqsubseteq, \sqcup, \bot)$. Then, lfp f exists, and the following holds.

$$\text{lfp } f = \sqcup \{ f^i(\bot) \mid i \in \mathbb{N} \}$$

While Tarski's theorem requires the operator to be monotonic, Kleene's fixpoint theorem requires the operator to be continuous, which is a strictly stronger hypothesis. On the other hand, Kleene's theorem only requires a CPO, while Tarski's theorem requires a complete lattice. Furthermore, Kleene's theorem gives a *constructive* definition for the least fixpoint of a continuous operator f as the limit of an iteration. While the characterization is constructive, this does not directly lead to an algorithm to compute the least fixpoint of f, as lfp f could be obtained after an infinite number of iterations.

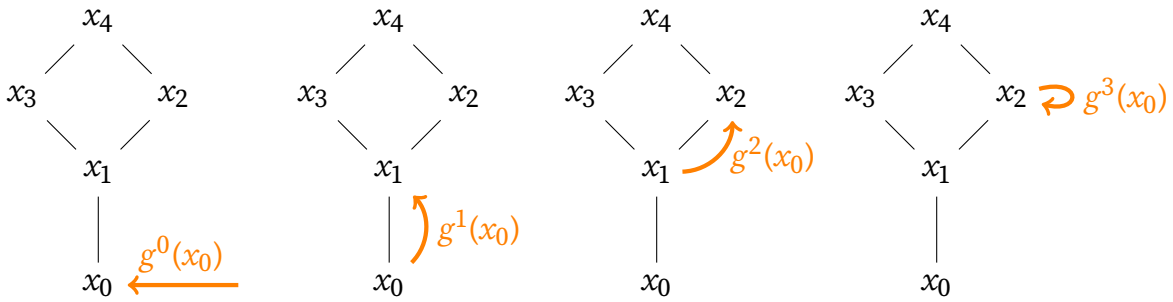


FIGURE 2.4. Kleene's fixpoint iterates

Example 2.8 (Kleene's least fixpoint computation)
Consider the CPO represented in Figure 2.4. We define the operator g as follows: $\{(x_0, x_1), (x_1, x_2), (x_2, x_2), (x_3, x_4), (x_4, x_4)\}$, and we observe that g is a continuous function. Figure 2.4 represents the first three Kleene's iterates for the function g. We observe that for all $i \geq 3$, $g^i(x_0) = x_2$. By applying Kleene's theorem, the following holds:

$$\operatorname{lfp} g = \sqcup \{ g^i(x_0) \mid i \in \mathbb{N} \} = x_2$$

The theory of fixpoints that we presented in this section is widely used in various areas of computer science. For instance, Stephen Kleene introduced the *star operator* of regular expressions (studied in Chapter 3), which can be expressed as an infinite union of languages, or equivalently as the fixpoint of a continuous operator over languages. Its existence is guaranteed by Kleene's theorem. Furthermore, both Kleene's and Tarski's fixpoint theorems are fundamental to define the semantics of programming languages. In fact, the behaviour of programming languages is formally defined as the fixpoint of a transition function over sets of program states. In Chapter 6 we formalize the behaviour of a simple programming language called WHILE as the least fixpoint of the transfer function, and the existence of such a fixpoint is guaranteed by the theorems presented in this section.

Part II

Verification of Security Properties for Regular Expressions

Chapter 3

Regular Expressions and Automata

In this chapter, we study a class of formal languages known as *regular languages*. We start by giving a lightweight introduction to formal languages in Section 3.1, and then we introduce regular expressions (Section 3.2) and finite automata (Section 3.3).

3.1. Formal languages

In mathematics, computer science, and linguistics, a *formal language* consists of words whose letters are taken from an alphabet Σ and are well-formed according to a specific set of rules. The alphabet Σ of a formal language consists of symbols that concatenate into strings of the language. Each string concatenated from symbols of this alphabet is called a *word*, and the words that belong to a particular formal language are sometimes called well-formed words or well-formed formulas. A formal language is often defined by means of a formal grammar, such as a regular grammar or context-free grammar, which consists of its formation rules. We now describe these concepts more formally.

Let Σ be a finite set of symbols. A finite sequence of elements of Σ is called a *finite word*. We denote the sequence $(a_1, a_2, \ldots, a_n)$ by mere juxtaposition:

$$a_1 a_2 \ldots a_n$$

The set of words is endowed with the operation of *concatenation product*, which associates to two words $u = a_1 \dots a_n$ and $v = b_1 \dots b_m$ the word $uv = a_1 \dots a_n b_1 \dots b_m$. We denote with ϵ the *empty word*. We denote by Σ^* the set of words on Σ and by Σ^+ the set of nonempty words, that is $\Sigma^+ \triangleq \Sigma^* \setminus \{\epsilon\}$. A *formal language*, or simply a *language*, is a subset of Σ^*. In what follows, we describe in detail the class of languages known as *regular*. There are many classes of languages other than the regular ones, such as *context-free* languages, *context-sensitive* languages, and *recursively enumerable* languages [58, 59, 60]. In this work, we focus primarily on regular languages and do not extensively discuss the others.

There are different equivalent ways to define regular languages. For instance, they can be defined as the set of languages recognized by *regular expressions* [58]. An alternative (but, as we will show, equivalent) definition is that regular languages are the class of languages recognized by *finite automata* (FAs). In the following sections, we formally introduce these concepts.

3.2. Regular expressions

Regular expressions (sometimes referred as regexes) are defined as follows.

$$\mathcal{R} \in \mathbb{R} \qquad \text{(Regular expressions)}$$
$$\mathcal{R} := \epsilon \mid a \in \Sigma \mid (\mathcal{R}_1 \mid \mathcal{R}_2) \mid \mathcal{R}_1 \cdot \mathcal{R}_2 \text{ (or } \mathcal{R}_1\mathcal{R}_2) \mid \mathcal{R}_1^*$$

We extend regular expressions with the possibility to recognize the *empty language*, namely the empty set of words, as follows.

$$\mathcal{R} \in \mathbb{R}^\perp \qquad \text{(Empty regular expressions)}$$
$$\mathcal{R} := \epsilon \mid a \in \Sigma \mid (\mathcal{R}_1 \mid \mathcal{R}_2) \mid \mathcal{R}_1 \cdot \mathcal{R}_2 \text{ (or } \mathcal{R}_1\mathcal{R}_2) \mid \mathcal{R}_1^* \mid \perp_r$$

Observe that $\mathbb{R} \subset \mathbb{R}^\perp$. Separating $\mathbb{R}$ from $\mathbb{R}^\perp$ is useful as empty regular expressions are not usually implemented in real-world programming languages. Nevertheless, $\mathbb{R}^\perp$ is important from a mathematical point of view, e.g., to make regular expressions closed under intersection. Let $a \in \Sigma$. We define the *language recognized* (or

accepted) by a regular expression $\mathcal{R} \in \mathbb{R}^{\perp}$ as follows.

$$\mathcal{L}(\perp_r) \triangleq \emptyset \tag{3.1}$$
$$\mathcal{L}(\epsilon) \triangleq \{\epsilon\} \tag{3.2}$$
$$\mathcal{L}(a) \triangleq \{a\} \tag{3.3}$$
$$\mathcal{L}(\mathcal{R}_1 \mid \mathcal{R}_2) \triangleq \mathcal{L}(\mathcal{R}_1) \cup \mathcal{L}(\mathcal{R}_2) \tag{3.4}$$
$$\mathcal{L}(\mathcal{R}_1 \cdot \mathcal{R}_2) \triangleq \mathcal{L}(\mathcal{R}_1) \cdot \mathcal{L}(\mathcal{R}_2) \tag{3.5}$$
$$\mathcal{L}(\mathcal{R}_1^*) \triangleq \bigcup_{i \geq 0} \mathcal{L}(\mathcal{R}_1)^i \tag{3.6}$$

Example 3.1 (Regular expression)
The regular expression aba^* recognizes the language $\{\, aba^n \mid n \geq 0 \,\}$.

When two regular expressions $\mathcal{R}_1$ and $\mathcal{R}_2$ recognize the same language, we write $\mathcal{R}_1 =_{\mathcal{L}} \mathcal{R}_2$. We can finally give the formal definition of *regular languages*, that is the class of languages recognized by regular expressions.

Definition 3.1 (Regular language)
Let $L \in \wp(\Sigma^*)$.

$$L \text{ is } \textit{regular} \overset{\triangle}{\iff} \exists \mathcal{R} \in \mathbb{R}^{\perp} : \mathcal{L}(\mathcal{R}) = L$$

It is sometimes useful to extend regular expressions with complement and intersection constructors as follows.

$\mathcal{R} \in \mathbb{R}^{+}$ (Extended regular expressions)
$\mathcal{R} := \epsilon \mid a \in \Sigma \mid (\mathcal{R}_1 \mid \mathcal{R}_2) \mid \mathcal{R}_1 \cdot \mathcal{R}_2$ (or $\mathcal{R}_1\mathcal{R}_2$) $\mid \mathcal{R}_1^* \mid \perp_r \mid \mathcal{R}_1 \cap \mathcal{R}_2 \mid \overline{\mathcal{R}_1}$

The last two constructors respectively correspond to the intersection and the complement. They recognize the following language:

$$\mathcal{L}(\mathcal{R}_1 \cap \mathcal{R}_2) \triangleq \mathcal{L}(\mathcal{R}_1) \cap \mathcal{L}(\mathcal{R}_2) \tag{3.7}$$
$$\mathcal{L}(\overline{\mathcal{R}_1}) \triangleq \Sigma^* \setminus \mathcal{L}(\mathcal{R}_1) \tag{3.8}$$

The new constructors do not increase the expressiveness of regular expressions,

as *regular languages are closed under complement and intersection* [58]. This implies that by intersecting two regular languages or complementing a regular language, we still obtain a regular language. While extended regular expressions are exactly as powerful as traditional regular expressions, they have been used in the literature to put forward efficient algorithms to solve the language inclusion problem [61, 62] (i.e., deciding whether $\mathcal{L}(\mathcal{R}_1) \subseteq \mathcal{L}(\mathcal{R}_2)$ holds). In Chapter 5, we describe how we take advantage of extended regular expressions to implement an efficient static analyzer for Regular Expression Denial of Service (ReDoS) vulnerabilities [36].

Smart constructors [61, 62] can be used to reduce the size of regular expressions. They are constructors that simplify the resulting regular expression when possible. For instance, consider the alternative definition of the concatenation of two regular expressions that ignores ϵ.

$$\texttt{concat}(\mathcal{R}_1, \mathcal{R}_2) \triangleq \begin{cases} \mathcal{R}_1 & \text{if } \mathcal{R}_2 = \epsilon \\ \mathcal{R}_2 & \text{if } \mathcal{R}_1 = \epsilon \\ \mathcal{R}_1 \cdot \mathcal{R}_2 & \text{otherwise} \end{cases} \tag{3.9}$$

The fundamental property that smart constructors must respect is the *language preservation,* meaning that regular expressions built with smart and regular constructors must accept exactly the same language.

Some constructors that are commonly used for regular expressions in programming languages are simply syntactic sugar for other basic constructors. Here we present some widely-used ones.

$$\mathcal{R}^? \triangleq \mathcal{R} \mid \epsilon \tag{3.10}$$

$$\mathcal{R}^+ \triangleq \mathcal{R}\mathcal{R}^* \tag{3.11}$$

$$[a - z] \triangleq a \mid b \mid \cdots \mid z \tag{3.12}$$

$$[A - Z] \triangleq A \mid B \mid \cdots \mid Z \tag{3.13}$$

$$[a - zA - Z] \triangleq [a - z] \mid [A - Z] \tag{3.14}$$

$$\backslash d \triangleq 0 \mid 1 \mid \cdots \mid 9 \tag{3.15}$$

$$\backslash w \triangleq \backslash d \mid [a - zA - Z] \mid \_ \tag{3.16}$$

These constructors do not increase the expressiveness of regular expressions, but

they are commonly used in real-world programs as they improve the readability of the patterns.

The *language membership problem* consists in deciding whether a word belongs to a language, namely deciding whether $w \in \mathcal{L}(\mathcal{R})$ holds for $w \in \Sigma^*$ and $\mathcal{R} \in \mathbb{R}^+$. The problem can be solved in linear time with respect to the length of the word [58], and we further discuss the time complexity in Section 3.3. Regular expression *derivatives* can be used to solve the language membership problem. In general, given a symbol a, the derivative of a regular expression $\mathcal{R}$ with respect to a is a regular expression that recognizes only those suffixes of strings with a leading a accepted by $\mathcal{R}$. Before formally defining derivatives, we define a helper function $\nu : \mathbb{R}^+ \to \mathbb{R}^+$ that has the following property:

$$\nu(\mathcal{R}) = \begin{cases} \epsilon & \text{if } \epsilon \in \mathcal{L}(\mathcal{R}) \\ \bot_r & \text{otherwise} \end{cases} \tag{3.17}$$

The function is defined as follows.

$$\nu(\epsilon) \triangleq \epsilon \tag{3.18}$$
$$\nu(a) \triangleq \bot_r \tag{3.19}$$
$$\nu(\bot_r) \triangleq \bot_r \tag{3.20}$$
$$\nu(\mathcal{R}_1 \mid \mathcal{R}_2) \triangleq \nu(\mathcal{R}_1) \mid \nu(\mathcal{R}_2) \tag{3.21}$$
$$\nu(\mathcal{R}_1 \cdot \mathcal{R}_2) \triangleq \nu(\mathcal{R}_1) \cap \nu(\mathcal{R}_2) \tag{3.22}$$
$$\nu(\mathcal{R}_1^*) \triangleq \epsilon \tag{3.23}$$
$$\nu(\mathcal{R}_1 \cap \mathcal{R}_2) \triangleq \nu(\mathcal{R}_1) \cap \nu(\mathcal{R}_2) \tag{3.24}$$
$$\nu(\overline{\mathcal{R}_1}) \triangleq \begin{cases} \epsilon & \text{if } \nu(\mathcal{R}_1) = \bot_r \\ \bot_r & \text{if } \nu(\mathcal{R}_1) = \epsilon \end{cases} \tag{3.25}$$

Let $a, b \in \Sigma$ such that $a \neq b$. We define the *Brzozowski's derivative* [63] $\partial_a : \mathbb{R}^+ \to \mathbb{R}^+$ with respect to the symbol a as follows.

$$\partial_a(\epsilon) \triangleq \bot_r \tag{3.26}$$
$$\partial_a(a) \triangleq \epsilon \tag{3.27}$$

$$\partial_a(b) \triangleq \bot_r \tag{3.28}$$
$$\partial_a(\bot_r) \triangleq \bot_r \tag{3.29}$$
$$\partial_a(\mathcal{R}_1 \cdot \mathcal{R}_2) \triangleq \partial_a(\mathcal{R}_1) \cdot \mathcal{R}_2 \mid \nu(\mathcal{R}_1) \cdot \partial_a(\mathcal{R}_2) \tag{3.30}$$
$$\partial_a(\mathcal{R}_1 \mid \mathcal{R}_2) \triangleq \partial_a(\mathcal{R}_1) \mid \partial_a(\mathcal{R}_2) \tag{3.31}$$
$$\partial_a(\mathcal{R}_1^*) \triangleq \partial_a(\mathcal{R}_1) \cdot \mathcal{R}_1^* \tag{3.32}$$
$$\partial_a(\mathcal{R}_1 \cap \mathcal{R}_2) \triangleq \partial_a(\mathcal{R}_1) \cap \partial_a(\mathcal{R}_2) \tag{3.33}$$
$$\partial_a(\overline{\mathcal{R}_1}) \triangleq \overline{\partial_a(\mathcal{R}_1)} \tag{3.34}$$

Example 3.2 (Brzozowski's derivative)

$$\begin{aligned}
\partial_a((a \mid a)^*) = &\ \partial_a((a \mid a)) \cdot (a \mid a)^* \\
= &\ (\partial_a(a) \mid \partial_a(a)) \cdot (a \mid a)^* \\
= &\ (\epsilon \mid \epsilon) \cdot (a \mid a)^* \\
=_{\mathcal{L}} &\ (a \mid a)^*
\end{aligned}$$

The derivatives can be extended to words as follows. Let $w \in \Sigma^*$ and $a \in \Sigma$.

$$\partial_\epsilon(\mathcal{R}) \triangleq \mathcal{R} \tag{3.35}$$
$$\partial_{wa}(\mathcal{R}) \triangleq \partial_w(\partial_a(\mathcal{R})) \tag{3.36}$$

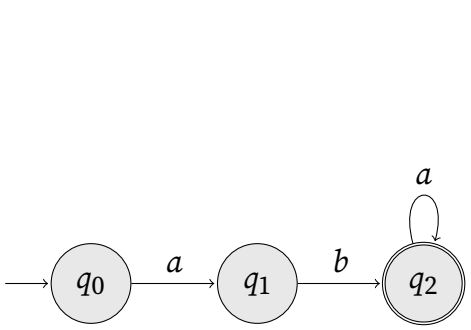
Theorem 3.1 (Brzozowski's theorem [63])
Let $w \in \Sigma^*$, $\mathcal{R} \in \mathbb{R}^+$.

$$w \in \mathcal{L}(\mathcal{R}) \iff \epsilon \in \mathcal{L}(\partial_w(\mathcal{R})) \iff \epsilon = \nu(\partial_w(\mathcal{R}))$$

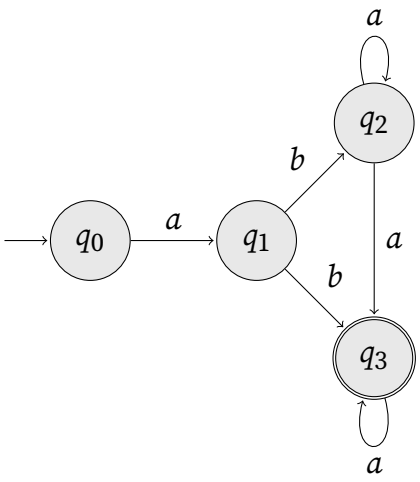
Thm. 3.1 trivially leads to an algorithm to solve the language membership problem for regular languages: first, compute the derivative of a regular expression with respect to a word w, and then check if ν applied to the result is exactly ϵ.

3.3. Finite automata

We now introduce *finite automata* (FAs), which we will show recognize exactly the same class of languages accepted by regular expressions. Automata can be



A. Example of a DFA



B. Example of a NFA

FIGURE 3.1. Examples of FAs

deterministic or *nondeterministic*. A *deterministic finite automaton* (DFA) is a tuple (Q, δ, i, F) where Q is a finite *set of states*, $\delta : Q \times \Sigma \to Q$ is the *transition function*, $i \in Q$ is the *initial state*, and $F \subseteq Q$ is the set of *final states*. A *nondeterministic finite automaton* (NFA) is a tuple (Q, δ, i, F) where Q is a finite *set of states*, $\delta \subseteq Q \times \Sigma \times Q$ is the *transition relation*, $i \in Q$ is the *initial state*, and $F \subseteq Q$ is the set of *final states*. Observe that the difference between a DFA and a NFA is that in the former δ is a function, while in the latter δ is a relation. We represent automata as graphs where the nodes correspond to the states, and the arcs between them are the transitions. The node with an incoming arc that does not originate from any other node is the initial state, while nodes with a double circle are the final states. Figures 3.1A-3.1B represent examples of FAs.

Let $q_0, q_1 \in Q$, $a \in \Sigma$. If $q_1 \in \delta(q_0, a)$, then we write $q_0 \xrightarrow{a} q_1$. Conversely, if $\nexists q_2 \in \delta(q_0, a)$, then we write $q_0 \xrightarrow{a}$. If $w \in \Sigma^*$, then $q_0 \overset{w}{\rightsquigarrow} q_1$ means that the state q_1 is reachable from q_0 by following the word w. More formally, by induction on the length of w: (i) if $w = \epsilon$ then $q_0 \overset{\epsilon}{\rightsquigarrow} q_1 \overset{\triangle}{\iff} q_0 = q_1$; (ii) if $w = av$ with $a \in \Sigma$, $v \in \Sigma^*$ then $q_0 \overset{av}{\rightsquigarrow} q_1 \overset{\triangle}{\iff} \exists q_2 \in \delta(q_0, a) : q_2 \overset{v}{\rightsquigarrow} q_1$. The *language accepted by the FA* $\mathcal{A} = (Q, \delta, i, F)$ is defined as follows.

$$\mathcal{L}(\mathcal{A}) \triangleq \{ w \in \Sigma^* \mid \exists q_f \in F : i \overset{w}{\rightsquigarrow} q_f \} \tag{3.37}$$

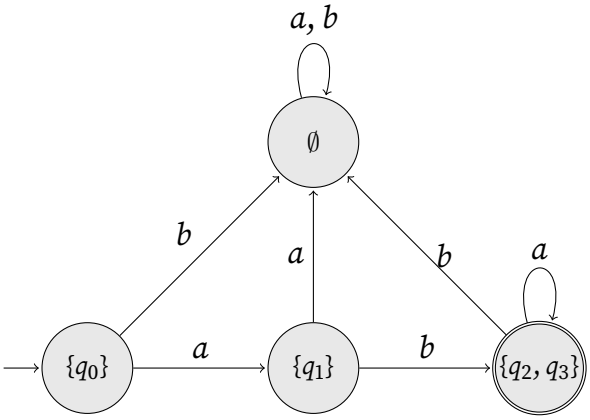


FIGURE 3.2. DFA obtained with the subset construction from the NFA in Figure 3.1B

Example 3.3 (Finite automata)
Figure 3.1A represents a DFA, while Figure 3.1B represents a NFA. The two automata recognize the same language, that is $\{ aba^n \mid n \geq 0 \}$.

When two automata $\mathcal{A}_1$ and $\mathcal{A}_2$ recognize the same language we write $\mathcal{A}_1 =_{\mathcal{L}} \mathcal{A}_2$. As it turns out, it is possible to convert a NFA into a DFA using a well-known technique called *subset construction* [58]. Let $\mathcal{N} = (Q_\mathcal{N}, \delta_\mathcal{N}, i_\mathcal{N}, F_\mathcal{N})$ be a NFA. We denote as $\mathtt{sub}(\mathcal{N}) \triangleq (Q_\mathcal{D}, \delta_\mathcal{D}, i_\mathcal{D}, F_\mathcal{D})$ the DFA obtained with the subset construction, where:

- $Q_\mathcal{D}$ is the *powerset* of $Q_\mathcal{N}$, namely $Q_\mathcal{D} \triangleq \wp(Q_\mathcal{N})$. Observe that if $Q_\mathcal{N}$ has n states, $Q_\mathcal{D}$ will have 2^n states. Often, many of these states are not reachable from the initial state, so that they can be omitted.

- $i_\mathcal{D} \triangleq \{ i_\mathcal{N} \}$

- $F_\mathcal{D} \triangleq \{ S \in Q_\mathcal{D} \mid S \cap F_\mathcal{N} \neq \emptyset \}$

- For each set $S \subseteq Q_\mathcal{N}$ and for each symbol $a \in \Sigma$:

$$\delta_\mathcal{D}(S, a) \triangleq \bigcup_{q \in S} \delta_\mathcal{N}(q, a)$$

Example 3.4 (Subset construction)
The automaton in Figure 3.2 is the DFA obtained by applying the subset construction to the NFA in Figure 3.1B. The unreachable states have been omitted.

The subset construction preserves the language recognized by the automaton, so that NFAs and DFAs recognize the same class of languages.

Theorem 3.2 (Correctness of `sub` [58])
Let $\mathcal{N}$ be a NFA.

$$\mathcal{L}(\mathcal{N}) = \mathcal{L}(\texttt{sub}(\mathcal{N}))$$

NFAs that accept ϵ as a valid symbol for the transition relation are known as *ϵ-NFAs*. As it turns out, it is possible to convert an ϵ-NFA into a DFA using a construction similar to the subset construction. We first define the ϵ-closure `eclose` of a state q_0 in an ϵ-NFA $\mathcal{E} = (Q, \delta, i, F)$ recursively as follows:

- **Base**: q_0 is in $\texttt{eclose}(q_0)$.
- **Induction**: If a state q_1 is in $\texttt{eclose}(q_0)$, and $q_2 \in \delta(q_1, \epsilon)$, then $q_2 \in \texttt{eclose}(q_0)$.

Let $\mathcal{E} = (Q_\mathcal{E}, \delta_\mathcal{E}, i_\mathcal{E}, F_\mathcal{E})$ be an ϵ-NFA. We denote as $\texttt{epsremove}(\mathcal{E}) \triangleq (Q_\mathcal{D}, \delta_\mathcal{D}, i_\mathcal{D}, F_\mathcal{D})$ the DFA where:

- All reachable states $Q_\mathcal{D}$ are ϵ-closed subsets of $Q_\mathcal{E}$, namely $Q_\mathcal{D}$ is the set of subsets $S \subseteq Q_\mathcal{E}$ such that $S = \texttt{eclose}(S)$.
- $i_\mathcal{D} \triangleq \texttt{eclose}(i_\mathcal{E})$
- $F_\mathcal{D} \triangleq \{ S \in Q_\mathcal{D} \mid S \cap F_\mathcal{E} \neq \emptyset \}$
- $\delta_\mathcal{D}(S, a)$ is computed, for all $a \in \Sigma$ and $S \in Q_\mathcal{D}$, as follows:
  - Let $S = \{ q_1, \ldots, q_n \}$
  - Let $\{ q'_1, \ldots, q'_m \} = \bigcup_{i=1}^{n} \delta_\mathcal{N}(q_i, a)$
  - Then, $\delta_\mathcal{D}(S, a) \triangleq \bigcup_{j=1}^{m} \texttt{eclose}(q'_j)$

The construction preserves the language of the ϵ-NFA, which implies that ϵ-NFAs and DFAs recognize the same class of languages.

Theorem 3.3 (Correctness of `epsremove` [58])
Let $\mathcal{E}$ be an ϵ-NFA.

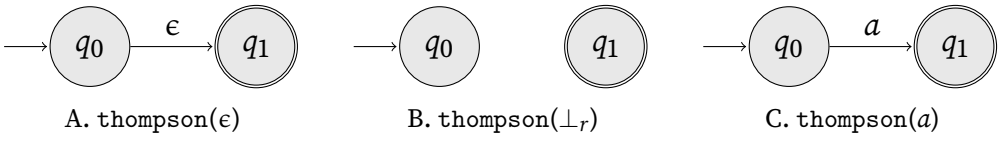
$$\mathcal{L}(\mathcal{E}) = \mathcal{L}(\texttt{epsremove}(\mathcal{E}))$$

NFAs with ϵ-transitions are commonly used when converting regular expressions into automata. The well-known *Thompson's construction* converts regular expressions into ϵ-NFAs [58]. This method is not only useful from a theoretical point of view, but it is used in real-world matching engines (such as RE2 [64]) to convert regular expressions into automata, which are then used to perform the matching [65]. The construction consists of a recursive procedure that builds an ϵ-NFA with exactly one final state. In Figure 3.3 we present the construction. The base cases in Figures 3.3A-3.3C are trivial, and the resulting automata have exactly two states. The inductive cases in Figures 3.3D-3.3F rely on the inductive construction for subexpressions. The left state in the boxes represents the initial state in the automaton built for a subexpression, while the right state is the final one. Figure 3.4 represents $\texttt{thompson}(aba^*)$. The Thompson construction preserves the language of the regular expressions.

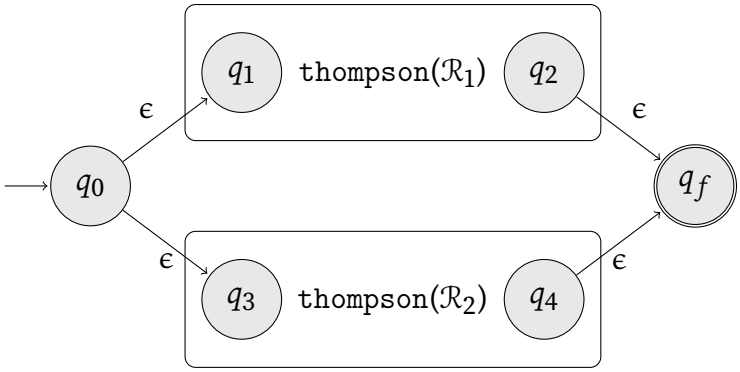
Theorem 3.4 (Correctness of `thompson` [58])
Let $\mathcal{R} \in \mathbb{R}^{\perp}$.

$$\mathcal{L}(\mathcal{R}) = \mathcal{L}(\texttt{thompson}(\mathcal{R}))$$

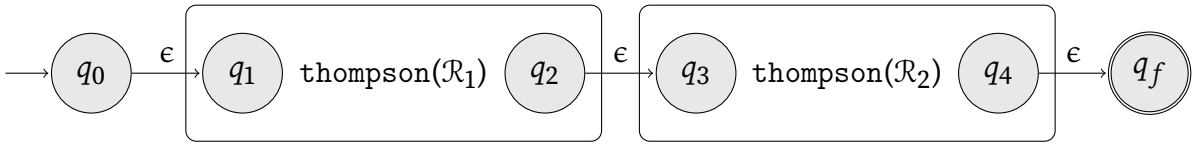
We showed that regular expressions can be converted into automata, so that to prove that the two recognize exactly the same class of languages it is sufficient to show that it is possible to convert automata into regular expressions while preserving the accepted language. In order to do this, we describe a method called *state elimination* [58]. The underlying idea is to interpret the transition labels in a DFA as regular expressions, and keep eliminating states until we have only the initial and final states. In this scenario, the language of the automaton is the union over all paths from the initial state to any final state of the language formed by concatenating the languages of the regular expressions along the path. Let q be the state that we want to remove, let $q_1, \ldots, q_n$ be its predecessors, and let $q'_1, \ldots, q'_m$ be its successors. We assume that q is not among its predecessors or successors. The transition from the predecessor q_i (for $i \in \{1, \ldots, n\}$) to q is denoted as $\mathcal{R}_i \in \mathbb{R}^{\perp}$, the



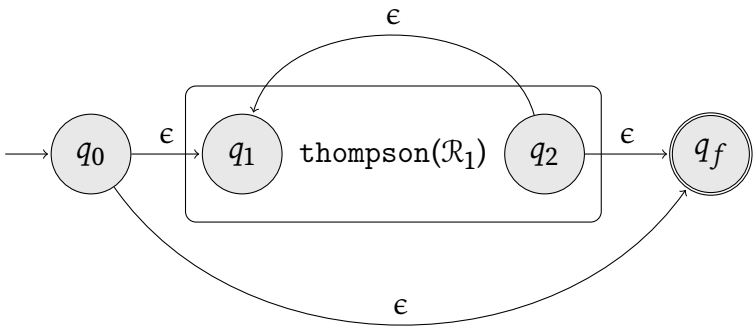
A. thompson(ϵ) B. thompson($\perp_r$) C. thompson(a)



D. thompson($\mathcal{R}_1 \mid \mathcal{R}_2$)



E. thompson($\mathcal{R}_1\mathcal{R}_2$)



F. thompson($\mathcal{R}^*$)

FIGURE 3.3. Thompson construction

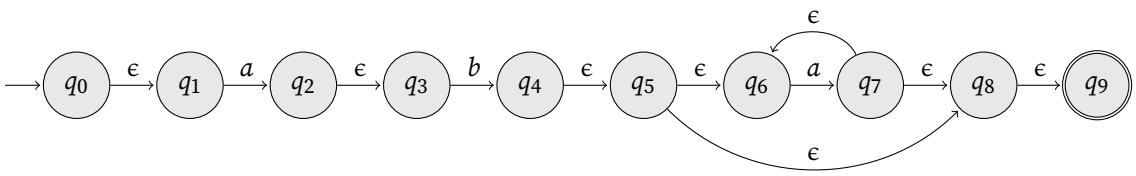
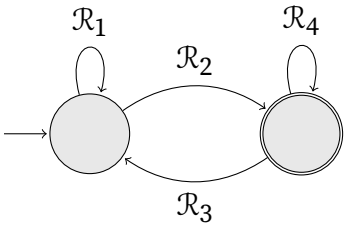
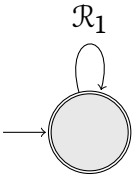


FIGURE 3.4. Thompson's automaton for aba^*



A. Generic two-states automaton



B. Generic single-state automaton

FIGURE 3.5. Generic shapes of automata with one or two states

transition from q to its successor q'_j (for $j \in \{1, \ldots, m\}$) is denoted as $\mathcal{R}'_j \in \mathbb{R}^{\perp}$, and the self loop from q to itself is denoted as $\mathcal{S} \in \mathbb{R}^{\perp}$. If q has no self-loop, $\mathcal{S}$ is simply $\perp_r$. We can remove q from the set of states by introducing, for each predecessor q_i and each successor q'_j a regular expression that represents all the paths that start from q_i, go through q, and reach q'_j. This regular expression is $\mathcal{R}_i\mathcal{S}^*\mathcal{R}'_j$, which is added, with the | operator, to the transition from q_i to q'_j. We can apply this elimination strategy to a DFA $\mathcal{D} = (Q, \delta, i, F)$ as follows:

- For each $q_f \in F$, eliminate all the states but q_f and i.
- If $q_f \neq i$, we obtain an automaton with only two states similar to the one in Figure 3.5A. The language accepted by the automaton is $(\mathcal{R}_1 \mid \mathcal{R}_2\mathcal{R}_4^*\mathcal{R}_3)^*\mathcal{R}_2\mathcal{R}_4^*$.
- If the start state is also accepting, we must perform a state elimination that gets rid of every state but i. In this case, we obtain a single-state automaton similar to the one in Figure 3.5B. The automaton accepts the language $\mathcal{R}_1^*$.
- The resulting regular expression is the union (using the | operator) of all the expressions derived with the described method.

Example 3.5 (State elimination construction)
aba^* is the regular expression resulting from applying the state elimination method to the automaton in Figure 3.1A.

We denote as $\texttt{stateelim}(\mathcal{D})$ the regular expression obtained by applying the state elimination procedure to a DFA $\mathcal{D}$. As it turns out, `stateelim` preserves the language recognized by the automaton.

Theorem 3.5 (Correctness of `stateelim` [58])
Let $\mathcal{D}$ be a DFA.

$$\mathcal{L}(\mathcal{D}) = \mathcal{L}(\texttt{stateelim}(\mathcal{D}))$$

We now present the last conversion method from regular expressions into automata that preserves the accepted language. *Glushkov's construction* [66] is a less-known algorithm to convert regular expressions into NFAs. Let $\mathcal{R} \in \mathbb{R}^{\perp}$ be a regular expression. We denote as $\texttt{glushkov}(\mathcal{R})$ the NFA built with the following steps:

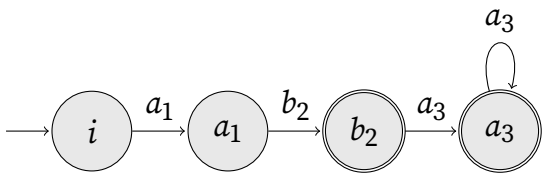
1. Each symbol in the regular expression is renamed so that each letter occurs at most once. This step is known as *linearization*, and we denote as Σ_1 the resulting alphabet and as $\mathcal{R}_1$ the linearized regular expression.

2. Compute the following sets by induction on the structure of the regular expression:

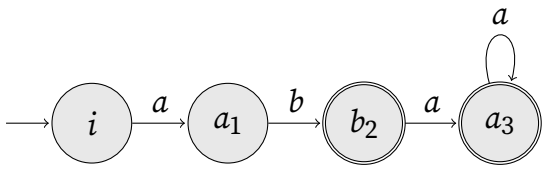
   - $P(\mathcal{R}_1) \triangleq \{\, a \in \Sigma_1 \mid a\Sigma_1^* \cap \mathcal{L}(\mathcal{R}_1) \neq \emptyset \,\}$
   - $D(\mathcal{R}_1) \triangleq \{\, a \in \Sigma_1 \mid \Sigma_1^* a \cap \mathcal{L}(\mathcal{R}_1) \neq \emptyset \,\}$
   - $F(\mathcal{R}_1) \triangleq \{\, ab \in \Sigma_1 \times \Sigma_1 \mid \Sigma_1^* ab \Sigma_1^* \cap \mathcal{L}(\mathcal{R}_1) \neq \emptyset \,\}$
   - $N(\mathcal{R}_1) \triangleq \{\, \epsilon \,\} \cap \mathcal{L}(\mathcal{R}_1)$

3. Construct a DFA $\mathcal{D} = (Q, \delta, i, F)$ as follows:

   - i is an artificial initial state.
   - Create a state for each symbol in the new alphabet, and to them add i: $Q \triangleq \Sigma_1 \cup \{\, i \,\}$.
   - For each $a \in P(\mathcal{R}_1)$, add a transition $i \xrightarrow{a} a$.



A. Glushkov's DFA for aba^*



B. Glushkov's NFA for aba^*

FIGURE 3.6. Example of Glushkov's automaton for aba^*

- For each $ab \in F(\mathcal{R}_1)$, add a transition $a \xrightarrow{b} b$.
- The set of final states is $D(\mathcal{R}_1)$. If $N(\mathcal{R}_1) = \{\epsilon\}$, then also i is final.

4. Build a NFA $\mathcal{N}$ by removing the linearization, replacing each symbol in Σ_1 with the original symbol in Σ.

Example 3.6 (Glushkov's contruction)
Consider the regular expression aba^*. The linearized regular expression is $a_1b_2a_3^*$, and the linearized alphabet is $\{a_1, b_2, a_3\}$. $P(a_1b_2a_3^*) = \{a_1\}$, $D(a_1b_2a_3^*) = \{b_2, a_3\}$, $F(a_1b_2a_3^*) = \{a_1b_2, b_2a_3, a_3a_3\}$, and $N(a_1b_2ba_3^*) = \emptyset$. Figure 3.6A represents the DFA obtained with the procedure described at the third step of the construction, while Figure 3.6B represents the resulting NFA for the Glushkov construction. Observe that in general automata built using the Glushkov construction present nondeterministic transitions, even though this is not the case in our example.

The Glushkov construction preserves the language of regular expressions.

Theorem 3.6 (Correctness of `glushkov` [66])
Let $\mathcal{R} \in \mathbb{R}^{\perp}$.

$$\mathcal{L}(\mathcal{R}) = \mathcal{L}(\texttt{glushkov}(\mathcal{R}))$$

Using the results presented in this section, we can finally prove that finite

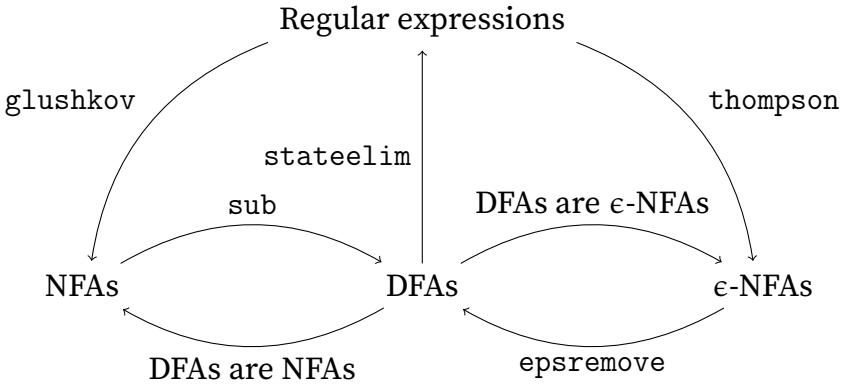


FIGURE 3.7. Language-preserving conversion methods between regular expressions and finite automata

automata and regular expressions recognize the same class of languages. Figure 3.7 summarizes the language-preserving conversion methods between regular expressions and finite automata.

Theorem 3.7 (Equivalance of automata and regular expressions)
Regular expressions, DFAs, NFAs, and ϵ-NFAs recognize the same class of languages.

We now describe a procedure to decide the language membership for DFAs, and we argue that it can be decided in linear time with respect to the length of the input word. Let $\mathcal{D} = (Q, \delta, i, F)$ be a DFA. The procedure $\texttt{member} : \Sigma^* \to Q \to \mathbb{B}$ returns $\texttt{tt}$ iff there is a path from the input state to a final state by matching the input word in $\mathcal{D}$. A word $w \in \Sigma^*$ is in $\mathcal{L}(\mathcal{D})$ iff $\texttt{member}(w, i) = \texttt{tt}$.

$$\texttt{member}(w, q) \triangleq \begin{cases} \texttt{tt} & \text{if } w = \epsilon \text{ and } q \in F \\ \texttt{ff} & \text{if } w = \epsilon \text{ and } q \notin F \\ \texttt{member}(v, q') & \text{if } w = av \text{ and } q \xrightarrow{a} q' \\ \texttt{ff} & \text{if } w = av \text{ and } q \xrightarrow{a} \end{cases} \tag{3.38}$$

The procedure considers the symbols in the input word: if there is a transition from the current state to another by matching the first symbol, then we perform a recursive call to $\texttt{member}$. If the input word is empty and q is a final state, then

the word is accepted by the automaton. Since the procedure performs at most $|w|$ recursive calls, it runs in $\mathcal{O}(|w|)$ time, so that deciding whether w is in $\mathcal{L}(\mathcal{D})$ is linear in the length of w. This shows that the language membership problem for regular languages can be decided in linear time with respect to the length of the word.

3.4. Conclusion

In this chapter, we formally defined regular languages as the class of languages recognized by regular expressions. After introducing DFAs, NFAs, and ϵ-NFAs, we argued that they all recognize the same class of languages, i.e., the regular languages. Lastly, we showed that the language membership problem for this class of languages can be decided in linear time in the length of the input word.

Chapter 4

Regular Expression Denial of Service Vulnerabilities Analysis

Modern programming languages often provide functions to manipulate regular expressions in standard libraries. If they offer support for *backreferences*, the matching algorithm has an exponential worst-case time complexity: for some so-called *vulnerable regular expressions*, an attacker can craft ad hoc strings to force the matcher to exhibit an exponential behaviour and perform a *Regular Expression Denial of Service* (ReDoS) attack. Interestingly, the vulnerable expressions do not even need to include backreferences in order to present exponential matching. In this chapter, we introduce a framework to prove the absence of this type of vulnerabilities in regular expressions. Our approach is *semantic*, meaning that it is rooted in a precise mathematical definition of the semantics of regular expression matching. The framework we propose is *sound*: once a regular expression is determined to be safe, then there is a strong mathematical guarantee about the absence of ReDoS vulnerabilities.

In Section 4.1, we introduce ReDoS attacks, and argue that they are a highly underestimated kind of Denial of Service attacks. Subsequently, in Section 4.2, we describe in detail why regular expression matching in real-world programming languages can generate such vulnerabilities despite the theoretical results from the last chapter showing that regular expression matching is linear time. In Section 4.3 and Section 4.4 we introduce a framework based on a *tree semantics* to statically

identify ReDoS vulnerabilities. In particular, we put forward an algorithm to extract an *overapproximation* of the set of words that are dangerous for a regular expression, effectively catching all possible attacks. In Section 4.5 we discuss possible extensions of this work, while in Section 4.6 we finally conclude by comparing our framework to other existing approaches to ReDoS detection. The proofs of the theoretical results are reported in Appendix A. This and the following chapter are based on a work published at TASE 2022 [51], and subsequently extended in Science of Computer Programming [52].

4.1. Motivation

Regular expressions are often used to verify that strings in programs match a given pattern. Modern programming languages offer support for regular expressions in standard libraries, and this encourages programmers to take advantage of them. However, matching engines in languages such as Python, JavaScript, and Java employ algorithms with exponential worst-case time complexity in the length of the string. This is because *backreferences* extend the expressiveness of regular expressions, and this comes at the cost of exponential matching in the worst case, even for regular expressions that do not exploit such features. An attacker can craft a string to force the matcher to exhibit the exponential behaviour to perform a ReDoS attack, a particular type of *algorithmic complexity attack* [35]. In Section 4.2 we describe why the cost of the matching can be exponential, and we give examples of some ReDoS vulnerabilities.

ReDoS attacks are vastly underestimated Denial of Service (DoS) attacks. In a recent study of regular expressions usage, in nearly 4,000 Python projects on Github, the authors find that over 42% of them contain regular expressions [67], while in [36] the authors find that 10% of the Node.js-based web services they examined are vulnerable to ReDoS. In this already harsh scenario, in [68] the authors find that only 38% of the developers that they surveyed knew about the *existence* of ReDoS attacks. Many well-known platforms observed such vulnerabilities in their systems: among them, we find Stack Overflow [37], Cloudflare [38], and iCloud [39]. Since it is difficult to detect ReDoS vulnerabilities with manual inspection, it is necessary to automate this process. However, for now, there is no practical and

widely adopted solution to detect ReDoS vulnerabilities.

There are many different approaches to static semantics-based ReDoS detection [69, 70, 71, 72], and they are all based on automata frameworks. Due to the difficulty in precisely modeling matching engines with automata, static analyzers often report both false positives and false negatives. In contrast, dynamic approaches to ReDoS detection [73] can hardly be used in practice, since performing dynamic testing on exponential algorithms can be excessively costly. Heuristics-based syntactical analyzer [74, 75, 76, 77] try to detect vulnerabilities by matching regular expressions against potentially dangerous patterns. However, these tools do not offer guarantees about the quality of the results, often reporting both false positives and false negatives.

In this chapter, we put forward a novel approach to statically detect ReDoS vulnerabilities. We get rid of the complexities to represent the behaviour of matching engines with automata by defining a *tree semantics* of the matching process. Next, we leverage it to introduce an analysis that determines whether a regular expression may be vulnerable or not. In particular, the analysis returns an *overapproximation* of the language of words that can cause exponential matching, being effectively *sound* but *not complete*. Nevertheless, our experiments (see Chapter 5) show that our approach reports a low number of false positives.

We focus on the most dangerous type of ReDoS vulnerability, namely when the matching is exponential. To successfully perform an attack that exploits superlinear but non-exponential matching, a malicious user must be allowed to insert very large strings. Such attacks are considerably less dangerous than the case that we consider.

Our technique not only eliminates the complexities related to using automata, but also allows extracting the language of possibly dangerous words, being strictly more expressive than most existing approaches. This expressiveness can be useful in different scenarios: for example, existing matching engines can use our algorithm to filter-out dangerous input strings. It is also possible to use the language of dangerous words by combining our framework with a string analysis in order to prove the absence of ReDoS vulnerabilities in real-world applications.

4.2. Background

4.2.1. Regular expression matching in programming languages

The majority of modern programming languages offer support for regular expression matching in their standard library. While language membership is well-known to be computable in linear time in the length of input strings for regular languages [58, 78] (see Section 3.2), matching engines designers often decide to increase the expressivity of regular expressions by introducing *backreferences* [79, 80], making the matching less efficient. Backreferences are constructs that match the same text as previously matched in a pattern. For instance, consider the Python regular expression `(a*)b\1`. The parentheses define a *capturing group*, namely a pattern that, once matched, will be available to be referenced again during the matching. The `\1` is a backreference that matches exactly the same sequence of characters matched in the first capturing group. As a result, the regular expression accepts the language $\{a^n b a^n \mid n \geq 0\}$, which is a non-regular *context-free language* [58]. Backreferences are radically different from other extensions of classic regular expressions that do not change the expressive power (see Section 3.2), as they cannot be converted into regular constructs. Matching engines that support backreferences use *backtracking algorithms*, which have exponential worst-time complexity. In fact, matching regular expressions with backreferences is known to be an NP-hard problem [81].

Lookaround assertions are features that enable users to specify assertions on the characters that will be matched (or have been matched) by a pattern. In case the assertion is not respected by the input string, the match fails. As it turns out, it is possible to express lookaround assertions by using only regular constructs, even though this can be very costly. In fact, the size of a deterministic finite automaton obtained from a regular expression with lookarounds can be, in the worst case, *doubly exponential* in the size of the regular expression [82, 83, 84]. For this reason, matching engines in mainstream languages support lookaround assertions with simple backtracking algorithms without resorting to automata, and this comes at the cost of exponential matching. Only recent advances showed that it is possible to implement an efficient matching engine that supports lookaround assertions by relying on *oracle regular expressions* [84]. However, these advancements have yet

TABLE 4.1. Matching algorithms comparison

Algorithm	**Complexity**	**Used in**
Finite automata [58, 78]	Linear	Rust [85, 86], RE2 Engine [64]
Backtracking [79, 80]	Exponential	Javascript (V8 runtime) [87, 88], Java [89, 90], PHP [91, 92], Perl [93, 94] Python [95, 96], Ruby [97, 98]

to be implemented in widely used matching engines. Consequently, mainstream programming languages still rely on inefficient backtracking-based techniques to support lookarounds.

Since in this work we target backtracking-based matching engines, in Section 4.2.4 we introduce the pseudocode for the backtracking matching procedure. As backreferences and lookarounds are, for the moment, not in the scope of our analysis, we present a simple version of the matching procedure that does not consider them. Observe that in Section 4.5 we outline some ideas for adding support for those advanced features.

While the great majority of programming languages allow backreferences and lookarounds, there exist some exceptions. For instance, Rust uses the techniques based on finite automata [58, 78] described in Section 3.3 to guarantee superior performance. This comes at the cost of forbidding backreferences and lookaround assertions. The official Rust documentation reports "Backreferences and arbitrary lookahead/lookbehind assertions are not provided. In return, regular expression searching provided by this package has excellent worst-case performance" [86]. In Table 4.1 we report the two main approaches to regular expression matching, their complexity with respect to the length of input strings, and some of the programming languages and matching engines that use them. Observe that there are also other approaches to regular expression matching, such as *derivatives-based matching* [61, 63, 99] (see Section 3.2), but they are not widely used in matching engines.

Observe that even if backreferences come at the cost of exponential matching, there are various scenarios in which the enhanced expressiveness is highly beneficial. An example is text editors, where programmers work on possibly large code bases, and being able to express advanced patterns can greatly improve the

```
1 import re
2 email_regex = r'^([0-9a-zA-Z]([-.\w]*[0-9a-zA-Z])*@(([0-9a-zA-Z
     ])+([-\w]*[0-9a-zA-Z])*\.)+[a-zA-Z]{2,9})$'
3 attack = 'a' * 50
4 re.match(email_regex, attack)
```

FIGURE 4.1. Python program that matches a dangerous string against a vulnerable regular expression

code editing speed. For instance, the popular Vim text editor [100] allows users to use backreferences to search and replace complex patterns. Another relevant case involves command line utilities, such as `sed`, that are specifically designed to search, replace, and delete patterns in large chunks of text. In such situations, the enhanced expressiveness greatly improves the capabilities and, therefore, the usefulness of the tool. Furthermore, backreferences are part of the POSIX standard for regular expressions [101], which highlights the fact that such features are well-established and unlikely to disappear from mainstream programming languages anytime soon.

4.2.2. ReDoS vulnerabilities

The majority of programming languages that offer support for regular expressions in standard libraries are vulnerable to ReDoS attacks. Figure 4.1 shows an example of a Python program that matches a string with a vulnerable regular expression that validates email addresses. The regular expression is taken from the Regexlib [102] database, and possibly many programmers used it. Executing the program on a modern computer with a 4GHz Intel Core i7-4790K CPU takes more than 24 hours. In Section 4.4, we give in-depth description of ReDoS vulnerabilities, but here we informally introduce why this behaviour arises. Consider the input string a^{50} and the subexpression `([-.\w]*[0-9a-zA-Z])*`: a can be matched in `[-.\w]*` or in `[0-9a-zA-Z]`. This implies that in `([-.\w]*[0-9a-zA-Z])*` there are four paths to match aa , eight for aaa, and in general 2^n for a^n. Normally, the matching engine accepts the first match, but here, as `@` does not appear in the string, it exhaustively explores all 2^{50} paths before concluding that no match is possible for a^{50} in the regular expression.

Most programming languages employ matching engines with exponential

worst-time complexity to support *backreferences* and *lookarounds* [79, 80]. Since our analysis is limited, for now, to classic regular expressions, we, as many other analyzers, do not support such features. Nevertheless, our approach is sufficient to analyze the great majority of regular expressions in real-world applications: in [67] the authors found that in nearly 4000 Python projects, only 4% of the regular expressions use lookarounds and up to 0.4% use backreferences. Yet, recent surveys determined that up to 10% of the web services they considered present ReDoS vulnerabilities [36]. This highlights how programmers use vulnerable matching engines while only occasionally taking advantage of advanced features, and motivates the need for a sound ReDoS analyzer even limited to classic regular expressions.

4.2.3. ReDoS detection

There are three main approaches to ReDoS detection:

Heuristics-based static detection. Heuristics-based static analyzers are tools that try to determine whether a regular expression is vulnerable or not using heuristics. They usually match the constructs of the input regular expression against a set of potentially dangerous patterns. For instance, SAFE-REGEX [75] checks that regular expressions do not present nested stars. By performing simple syntactic checks, these tools are typically faster than others. On the other hand, since they do not rely on formal semantics, they can report both false positives and false negatives, often resulting in low-quality outcomes. In our experimental evaluation in Chapter 5 , we demonstrate that these tools produce unsatisfactory results. The tools SAFE-REGEX [75], REGEXPLOIT [76], and REDOS-DETECTOR [77] are examples of heuristics-based static analyzers.

Semantics-based static detection. There are many approaches to semantics-based static ReDoS detection [69, 70, 71, 72], and they all rely on automata. In those frameworks, regular expressions are first transformed into automata, which are then analyzed to determine whether they are vulnerable or not. The main problem is that transforming regular expressions into automata can remove or inject vulnerabilities. This is often a source of both false positives and false negatives. We discuss semantics-based static analyzers based on automata in detail in Section 4.6, and we compare them to our approach which is also

semantics-based, but operates on regular expressions instead of automata.

Dynamic detection. A dynamic analyzer generates strings that are fed to the matching engine. Then, the tool measures the time for the matching and determines whether a regular expression is vulnerable or not. These tools are sensibly slower than static analyzers, because performing testing on exponential algorithms can be excessively time-consuming. While it is possible to configure generic fuzzers, such as SLOWFUZZ [103], to detect ReDoS vulnerabilities, in [73] the authors present RESCUE: a more precise gray-box approach which leverages a genetic algorithm to efficiently generate input strings.

As described in Chapter 5, in our experiments we find that heuristics-based static analyzers raise a sensibly higher number of false positives and false negatives compared to other approaches. Nevertheless, heuristics-based detectors are the mostly used tools in practice. For instance, SAFE-REGEX [75] averages from 18,000 to 20,000 downloads per week on NPM [104].

4.2.4. Backtracking regular expression matching

In this section, we provide the pseudocode of the matching procedure. While it is simple and concise, it models the concrete behaviour of realistic matching engines. The pseudocode ignores details specific to a particular implementation, giving a high-level description of the procedure. Our algorithm is a trivial adaptation of the one presented in [89], which models Java's matching engine. Classic textbooks about regular expressions [79, 80] confirm that matching engines in standard libraries employ a backtracking procedure for the matching. As backreferences and lookarounds are, for the moment, not in the scope of our analysis, we present a simple version of the matching procedure that does not consider them.

In what follows, we assume that regular expressions automatically remove ϵ in the concatenation (this is known as a *smart-constructor*, see Section 3.2), so that $\mathcal{R} \cdot \epsilon$ and $\epsilon \cdot \mathcal{R}$ are always simplified to $\mathcal{R}$. This allows representing regular expressions as they are implemented in programming languages, where ϵ cannot be inserted by the user in the concatenation. We define two functions to deconstruct the

Algorithm 1: Matching algorithm pseudocode

```
1  function match(R : ℝ, w : Σ*, C : ℘(ℝ)) → 𝔹
2      if R ∈ C then
3          return ff
4      switch (regex-head(R), regex-tail(R)) do
5          case (ϵ, ϵ) do
6              return w = ϵ
7          case (a, R1) do
8              if w = a w1 then return match(R1, w1, ∅)
9              else return ff
10         case (R1 | R2, R3) do
11             return match(R1 R3, w, C) ∨ match(R2 R3, w, C)
12         case (R1*, R2) do
13             return match(R1 R1* R2, w, C ∪ {R1* R2}) ∨ match(R2, w, C)
```

concatenation of a regular expression $\mathcal{R}$.

$$\texttt{regex-head}(\mathcal{R}) \triangleq \begin{cases} \texttt{regex-head}(\mathcal{R}_1) & \text{if } \mathcal{R} = \mathcal{R}_1\mathcal{R}_2 \\ \mathcal{R} & \text{otherwise} \end{cases} \tag{4.1}$$

$$\texttt{regex-tail}(\mathcal{R}) \triangleq \begin{cases} \texttt{regex-tail}(\mathcal{R}_1) \cdot \mathcal{R}_2 & \text{if } \mathcal{R} = \mathcal{R}_1\mathcal{R}_2 \\ \epsilon & \text{otherwise} \end{cases} \tag{4.2}$$

We define the function $\texttt{nstars} : \mathbb{R} \to \mathbb{N}$ that returns the number of stars in a regular expression as follows.

$$\texttt{nstars}(\mathcal{R}) \triangleq \begin{cases} 0 & \text{if } \mathcal{R} = a \text{ or } \mathcal{R} = \epsilon \\ \texttt{nstars}(\mathcal{R}_1) + \texttt{nstars}(\mathcal{R}_2) & \text{if } \mathcal{R} = \mathcal{R}_1\mathcal{R}_2 \text{ or } \mathcal{R} = \mathcal{R}_1 \mid \mathcal{R}_2 \\ 1 + \texttt{nstars}(\mathcal{R}_1) & \text{if } \mathcal{R} = \mathcal{R}_1^* \end{cases} \tag{4.3}$$

In Algorithm 1, we present the matching procedure. The logic operators are short-circuit: as soon as the input word is matched, the unexplored branches of the regular expression are not considered. The behaviour of function `match` depends on the first constructor in the concatenation of the regular expression, and the remaining portion can possibly be ϵ. The algorithm is rather trivial, but it models two important aspects of matching engines. First, it implements a *prioritization*

mechanism that: (1) tries to expand the left branch before the right branch in alternatives; (2) tries to match as many characters as possible in the body of the stars. Second, the algorithm prevents infinite ϵ-matching loops. Consider $(\epsilon \mid a)^*$: if we remove line 3, the procedure keeps expanding the body of the star forever, never consuming any character of the input string. To prevent this, when a star is expanded, it is inserted in C, that is the set of stars that cannot be expanded again. Initially, C must be the empty set. The stars are removed from C only when at least one character is matched.

Observe that usually in matching engines the match is successful even if just a prefix of the word matches the regular expression [79, 80], and this is known as *submatch* or *partial match* semantics. We can easily model this behaviour by appending Σ^* at the end of regular expressions [70]. For the sake of simplicity and without loss of generality, we assume that the match is successful only if the entire word is matched (*fullmatch* semantics). Since real-world matching engines use the partial match semantics, our implementation of the analysis (see Chapter 5) assumes instead by default such semantics. The translation between the two simply rewrites $\mathcal{R} \in \mathbb{R}$ as $\mathcal{R}\Sigma^*$.

4.3. Regular expression matching semantics

In this section, we first define a small-step operational semantics as a transition relation between the configurations of the matching engine. We then use it to put forward a tree semantics that precisely describes the steps performed during the matching. Lastly, we use the semantics to formally define ReDoS vulnerabilities.

We extend $\mathbb{R}$ to represent whether a star has been expanded and not a single character has been matched yet. The syntax of a regular expression $\mathcal{R} \in \mathbb{R}^{\mathcal{T}}$ is given by the following grammar.

$$\mathcal{R} \in \mathbb{R}^{\mathcal{T}} \qquad \text{(Transitional regular expressions)}$$
$$\mathcal{R} := \epsilon \mid a \in \Sigma \mid (\mathcal{R}_1 \mid \mathcal{R}_2) \mid \mathcal{R}_1 \cdot \mathcal{R}_2 \text{ (or } \mathcal{R}_1\mathcal{R}_2) \mid \mathcal{R}_1^* \mid \mathcal{R}_1^{\overline{*}}$$

It differs from traditional regular expressions for the *closed star*, namely $\mathcal{R}^{\overline{*}}$. It is a star that cannot be expanded again in order to prevent infinite ϵ-matching loops. We will formalize this concept with the transition relation. The closed stars avoid

the necessity to keep a separate set of expressions (C in Algorithm 1) during the matching: the information is implicitly included in the regular expression.

We call a pair in $\mathbb{R}^{\mathcal{T}} \times \Sigma^* \triangleq \mathbb{S}_r$ a *matching engine state,* and it describes the configuration of the matching engine. The first component is the regular expression that the matcher is expanding, and the second is the suffix of the input word that still has to be matched. We define the function $\texttt{refresh} : \mathbb{R}^{\mathcal{T}} \to \mathbb{R}$ to transform the closed stars back into regular stars as follows.

$$\texttt{refresh}(\epsilon) \triangleq \epsilon \tag{4.4}$$
$$\texttt{refresh}(a) \triangleq a \tag{4.5}$$
$$\texttt{refresh}(\mathcal{R}_1 \mid \mathcal{R}_2) \triangleq \texttt{refresh}(\mathcal{R}_1) \mid \texttt{refresh}(\mathcal{R}_2) \tag{4.6}$$
$$\texttt{refresh}(\mathcal{R}_1 \cdot \mathcal{R}_2) \triangleq \texttt{refresh}(\mathcal{R}_1) \cdot \texttt{refresh}(\mathcal{R}_2) \tag{4.7}$$
$$\texttt{refresh}(\mathcal{R}_1^*) \triangleq \texttt{refresh}(\mathcal{R}_1)^* \tag{4.8}$$
$$\texttt{refresh}(\mathcal{R}_1^{\overline{*}}) \triangleq \texttt{refresh}(\mathcal{R}_1)^* \tag{4.9}$$

We define the set of *actions* as $\mathbb{A} \triangleq \{\text{◐}, \text{◑}, \circledast, \bigcirc\} \cup \{\circledcirc_a \mid a \in \Sigma\}$. The actions ◐ and ◑ denote when we expand, respectively, the left or the right branch of an alternative, while $\circledast$ and $\bigcirc$ correspond to expanding or ignoring a star construct. Finally, $\circledcirc_a$ represents the action to match a symbol a. Let $a \in \Sigma$ and $w \in \Sigma^*$. We can finally define the *transition relation* between states. It is not deterministic, but sequences of actions will be ordered later in this section.

$$(a, aw) \xrightarrow{\circledcirc_a} (\epsilon, w) \tag{4.10}$$
$$(a\mathcal{R}_1, aw) \xrightarrow{\circledcirc_a} (\texttt{refresh}(\mathcal{R}_1), w) \tag{4.11}$$
$$(\mathcal{R}_1 \mid \mathcal{R}_2, w) \xrightarrow{\text{◐}} (\mathcal{R}_1, w) \tag{4.12}$$
$$((\mathcal{R}_1 \mid \mathcal{R}_2)\mathcal{R}_3, w) \xrightarrow{\text{◐}} (\mathcal{R}_1\mathcal{R}_3, w) \tag{4.13}$$
$$(\mathcal{R}_1 \mid \mathcal{R}_2, w) \xrightarrow{\text{◑}} (\mathcal{R}_2, w) \tag{4.14}$$
$$((\mathcal{R}_1 \mid \mathcal{R}_2)\mathcal{R}_3, w) \xrightarrow{\text{◑}} (\mathcal{R}_2\mathcal{R}_3, w) \tag{4.15}$$
$$(\mathcal{R}_1^*, w) \xrightarrow{\circledast} (\mathcal{R}_1\mathcal{R}_1^{\overline{*}}, w) \tag{4.16}$$
$$(\mathcal{R}_1^*\mathcal{R}_2, w) \xrightarrow{\circledast} (\mathcal{R}_1\mathcal{R}_1^{\overline{*}}\mathcal{R}_2, w) \tag{4.17}$$
$$(\mathcal{R}_1^*, w) \xrightarrow{\bigcirc} (\epsilon, w) \tag{4.18}$$

$$(\mathcal{R}_1^* \mathcal{R}_2, w) \xrightarrow{\bigcirc} (\mathcal{R}_2, w) \tag{4.19}$$

The transition relation describes all possible choices of the matching engine according to the state. Observe that with the $\circledast$ action the star becomes $\overline{*}$, and it cannot be expanded again until a character is matched. In fact, the transition relation is not defined for $\mathcal{R}^{\overline{*}}$. After consuming a character of the input word, we apply the function `refresh` to mark all stars as expandable. Observe that the transition relation describes all possible actions that Algorithm 1 might perform in a particular state.

We now leverage the transition relation to define a tree semantics for the matching procedure. We begin by defining the set of *execution traces* for $(\mathcal{R}_0, w_0) \in \mathbb{S}_r$.

$$\mathcal{T}((\mathcal{R}_0, w_0)) \triangleq \{ (\mathcal{R}_0, w_0) \xrightarrow{\mathcal{A}_1} (\mathcal{R}_1, w_1) \xrightarrow{\mathcal{A}_2} \cdots \xrightarrow{\mathcal{A}_n} (\mathcal{R}_n, w_n) \mid \forall i \in [0, n-1] : \mathcal{A}_i \in \mathbb{A} \text{ and } (\mathcal{R}_i, w_i) \xrightarrow{\mathcal{A}_{i+1}} (\mathcal{R}_{i+1}, w_{i+1}) \} \tag{4.20}$$

Example 4.1 ($\mathcal{T}((a^*, a))$)

$$\begin{aligned}
\mathcal{T}((a^*, a)) = \{ & (a^*, a), \\
& (a^*, a) \xrightarrow{\circledast} (aa^{\overline{*}}, a), \\
& (a^*, a) \xrightarrow{\circledast} (aa^{\overline{*}}, a) \xrightarrow{\circledcirc a} (a^*, \epsilon), \\
& (a^*, a) \xrightarrow{\circledast} (aa^{\overline{*}}, a) \xrightarrow{\circledcirc a} (a^*, \epsilon) \xrightarrow{\bigcirc} (\epsilon, \epsilon), \\
& (a^*, a) \xrightarrow{\circledast} (aa^{\overline{*}}, a) \xrightarrow{\circledcirc a} (a^*, \epsilon) \xrightarrow{\circledast} (aa^{\overline{*}}, \epsilon), \\
& (a^*, a) \xrightarrow{\bigcirc} (\epsilon, a) \}
\end{aligned}$$

We denote the last state of a trace t as $\ell(t)$ and we define the set of *complete execution traces* as $\mathcal{T}_c((\mathcal{R}, w)) \triangleq \{ t \in \mathcal{T}((\mathcal{R}, w)) \mid \ell(t) \nrightarrow \}$. Observe that $\mathcal{T}_c((\mathcal{R}, w))$ represents all possible executions of the matching engine from $(\mathcal{R}, w)$ up to a state in which it is not possible to continue.

Example 4.2 ($\mathcal{T}_c((a^*, a))$)

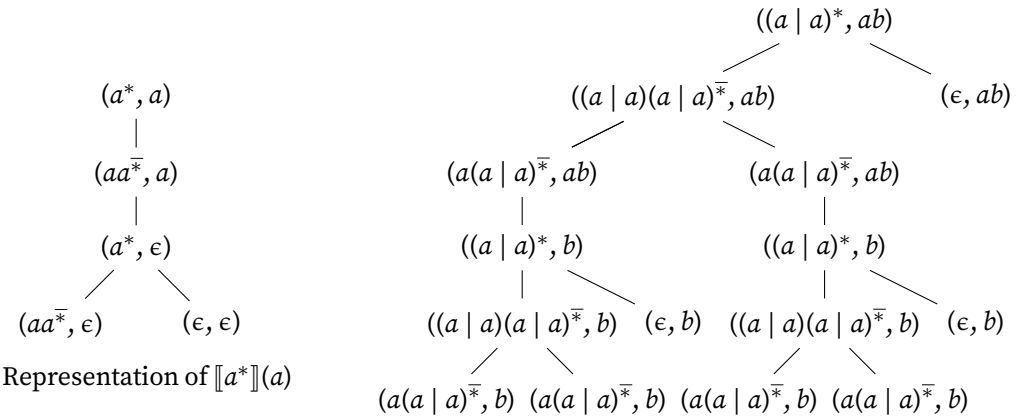
$$\begin{aligned}\mathcal{T}_c((a^*, a)) = \{\, &(a^*, a) \xrightarrow{\circledast} (aa^{\overline{*}}, a) \xrightarrow{\circledcirc_a} (a^*, \epsilon) \xrightarrow{\bigcirc} (\epsilon, \epsilon),\\ &(a^*, a) \xrightarrow{\circledast} (aa^{\overline{*}}, a) \xrightarrow{\circledcirc_a} (a^*, \epsilon) \xrightarrow{\circledast} (aa^{\overline{*}}, \epsilon),\\ &(a^*, a) \xrightarrow{\bigcirc} (\epsilon, a)\,\}\end{aligned}$$

We say that two traces are part of the same matching run if they have the same initial state. To build the matching tree, we need to order the traces from the first that will be explored to the last. Let t_1, t_2 be two complete execution traces in the same matching run, and let $(\mathcal{R}_1, w_1)$ be the last state in the longest common prefix between t_1 and t_2. We impose a lexical order $\sqsubseteq$ such that $t_1 \sqsubseteq t_2$ iff the action chosen by t_1 after $(\mathcal{R}_1, w_1)$ is either $\CIRCLE$ or $\circledast$. This order assigns higher priority to the traces that choose to expand the left branch of the alternative or to expand the body of the star, which is the standard behaviour of matching engines. Let T be a set of complete execution traces such that all of them are part of the same matching run. We denote with $\mathcal{O}_{\sqsubseteq}(T)$ the sequence of traces in T ordered by $\sqsubseteq$.

Example 4.3 ($(\mathcal{O}_{\sqsubseteq} \circ \mathcal{T}_c)((a^*, a))$)

$$\begin{aligned}(\mathcal{O}_{\sqsubseteq} \circ \mathcal{T}_c)((a^*, a)) = &(a^*, a) \xrightarrow{\circledast} (aa^{\overline{*}}, a) \xrightarrow{\circledcirc_a} (a^*, \epsilon) \xrightarrow{\circledast} (aa^{\overline{*}}, \epsilon),\\ &(a^*, a) \xrightarrow{\circledast} (aa^{\overline{*}}, a) \xrightarrow{\circledcirc_a} (a^*, \epsilon) \xrightarrow{\bigcirc} (\epsilon, \epsilon),\\ &(a^*, a) \xrightarrow{\bigcirc} (\epsilon, a)\end{aligned}$$

Observe that $(\mathcal{O}_{\sqsubseteq} \circ \mathcal{T}_c)((\mathcal{R}, w))$ corresponds to the ordered sequence of *all* complete execution traces. During the concrete execution, some of them will never be explored, because as soon as the state (ϵ, ϵ) is found, the procedure terminates. We want to remove from $(\mathcal{O}_{\sqsubseteq} \circ \mathcal{T}_c)((\mathcal{R}, w))$ those traces that appear after (ϵ, ϵ). Let $S = t_1, t_2, \ldots, t_n$ be a sequence of complete execution traces. We denote by $\mathcal{F}_\epsilon(S)$ the sequence $t_1, t_2, \ldots, t_k$ such that t_k is the first trace for which it holds that $\ell(t_k) = (\epsilon, \epsilon)$. If there is no such trace, then $k = n$ (i.e., there is an exhaustive exploration of all the traces before failing).



A. Representation of $[\![a^*]\!](a)$

B. Representation of $[\![(a \mid a)^*]\!](ab)$

FIGURE 4.2. Examples of matching trees

Example 4.4 $((\mathcal{F}_\epsilon \circ \mathcal{O}_\sqsubseteq \circ \mathcal{T}_c)((a^*, a)))$

$$(\mathcal{F}_\epsilon \circ \mathcal{O}_\sqsubseteq \circ \mathcal{T}_c)((a^*, a)) = (a^*, a) \xrightarrow{\circledast} (aa^{\overline{*}}, a) \xrightarrow{\circledcirc_a} (a^*, \epsilon) \xrightarrow{\circledast} (aa^{\overline{*}}, \epsilon),$$
$$(a^*, a) \xrightarrow{\circledast} (aa^{\overline{*}}, a) \xrightarrow{\circledcirc_a} (a^*, \epsilon) \xrightarrow{\bigcirc} (\epsilon, \epsilon)$$

Let S be a sequence of complete execution traces such that all of them are part of the same matching run. We denote by $\curlyvee(S)$ the tree obtained by merging the common prefixes in S.

Definition 4.1 (Matching tree semantics)
Let $\mathcal{R} \in \mathbb{R}^{\mathcal{T}}$ and $w \in \Sigma^*$. The *matching tree semantics* of $\mathcal{R}$ with respect to w is given by the following tree.

$$[\![\mathcal{R}]\!](w) \triangleq (\curlyvee \circ \mathcal{F}_\epsilon \circ \mathcal{O}_\sqsubseteq \circ \mathcal{T}_c)((\mathcal{R}, w))$$

Example 4.5 (Matching tree)
Figure 4.2 represents some examples of matching trees. Figure 4.2A represents a tree in which the matching is successful, while in Figure 4.2B the matching fails. One can reconstruct the steps carried out by the matching engine by doing a depth-first left-to-right traversal of the semantic tree.

We denote the number of nodes in a tree t with $|t|$ and its set of leaves as

$\texttt{leaves}(t)$. We define the *language recognized by* $\mathcal{R} \in \mathbb{R}^{\mathcal{T}}$ as follows.

$$\mathcal{L}(\mathcal{R}) \triangleq \{\, w \in \Sigma^* \mid (\epsilon, \epsilon) \in \texttt{leaves}([\![\mathcal{R}]\!](w)) \,\} \tag{4.21}$$

We now give the definition of ReDoS vulnerability, using the one that firstly appeared in [71], but adapted to our semantics.

Definition 4.2 (ReDoS Vulnerability)
Let $\mathcal{R} \in \mathbb{R}$ and $n \in \mathbb{N}$.

$$M_{\mathcal{R}}(n) \triangleq \max\{\, |[\![\mathcal{R}]\!](w)| \mid w \in \Sigma^*, |w| \leq n \,\}$$

We say that $\mathcal{R}$ has a *ReDoS vulnerability* iff $M_{\mathcal{R}} \in \Omega(2^n)$.

Observe that we characterize ReDoS vulnerabilities in terms of the matching trees' *size*, while in practice we are interested in the *execution time* for the matching procedure. It is sufficient to observe that matching engines explore, one at the time, all the states in the matching tree, so that there is a direct correspondence between the size of matching trees and the execution time for the matching procedure.

Example 4.6 (ReDoS vulnerability)
The regular expression $(a \mid a)^*$ presents a ReDoS vulnerability. Consider the word $a^n b$: the first a can be matched both in the left or in the right branch of the alternative. This implies that there are four possibilities to match the second a, namely by expanding left-left, left-right, right-left or right-right. In general, there are 2^n possible expansions to match a^n. All of them will be explored because the suffix b causes the match to fail, thereby forcing the engine to expand all of them.

The following lemma formalizes the intuition that the length of input words is an upper bound for the height of matching trees.

Lemma 4.1 (Height of matching tree)

Let $\mathcal{R} \in \mathbb{R}^{\mathcal{T}}$, $w \in \Sigma^*$, and h be the height of $[\![\mathcal{R}]\!](w)$.

$$h = \mathcal{O}(|w|)$$

In this section, we put forward a small-step operational semantics that formalizes the behaviour of matching engines. Similarly to trace semantics for programs, our semantics describes the behaviour of a regular expression in terms of a set of trees (one for each word). As we show in the next section, by relying on a precise mathematical description of the regular expression semantics, it is possible to reason on the behaviour of *all* possible matching runs. This opens the possibility to prove properties of the behaviour of regular expressions, similarly to how program analysis techniques prove properties of programs.

4.4. ReDoS vulnerabilities detection

In this section, we describe a framework to statically detect exponential ReDoS vulnerabilities. The analysis we propose derives from a regular expression an overapproximation of the set of dangerous words, namely those that can possibly cause an exponential ReDoS attack. The analysis is *sound* but not *complete*: any true vulnerability will be reported, but the algorithm can occasionally raise *false positives* (i.e., harmless regular expressions can be considered dangerous). Nevertheless, as discussed in Chapter 5, our experiments show that in practice our approach is precise and reports only 49 false positives over 74,669 regular expressions.

Intuitively, there is an exponential ReDoS vulnerability in a star if it is possible to match a word with at least two different traces. Consider $(a \mid a)^*$: a is matched in two traces by expanding the left or the right branch of the alternative. This implies that there are four traces to match aa, eight for aaa and in general 2^n for a^n. Nevertheless, $[\![(a \mid a)^*]\!](a^n)$ is not an exponential tree, because the match succeeds after expanding the left branch of the alternative n times. By appending a character that makes the match fail after a^n, an attacker can force the matching engine to explore all traces, effectively performing a ReDoS attack. This is the reason why $|[\![(a \mid a)^*]\!](a^n b)| = \Theta(2^n)$ (see Example 4.6).

First, we define a function $\mathcal{M}_2 : \mathbb{R}^{\mathcal{T}} \to \Sigma^*$ to extract the set of words that are

Algorithm 2: Compute $\mathcal{M}_2(\mathcal{R})$

```
1  function M2(R : ℝ) → ℝ^⊥
2      return M2-rec(R, ∅)
3  function M2-rec(R : ℝ^𝒯, E : ℘(ℝ^𝒯)) → ℝ^⊥
4      if R ∈ E then
5          return ⊥_r
6      switch (regex-head(R), regex-tail(R)) do
7          case (ϵ, ϵ) ∨ (R_1^*, R_2) do
8              return ⊥_r
9          case (a, R_1) do
10             return a · M2-rec(refresh(R_1), E)
11         case (R_1 | R_2, R_3) do
12             inter ← R_1R_3 ∩_∉ R_2R_3
13             left  ← M2-rec(R_1R_3, E)
14             right ← M2-rec(R_2R_3, E)
15             return inter | left | right
16         case (R_1^*, R_2) do
17             inter ← R_1R_1^*R_2 ∩_∉ R_2
18             left  ← M2-rec(R_1R_1^*R_2, E ∪ {R})
19             right ← R_1^* · M2-rec(R_2, E)
20             return inter | left | right
```

matched in at least two traces in a regular expression $\mathcal{R}$.

$$\mathcal{M}_2(\mathcal{R}) \triangleq \{\, w \in \Sigma^+ \mid \exists t_1, t_2 \in \mathcal{T}_c((\mathcal{R}, w)) : t_1 \neq t_2 \text{ and } \ell(t_1) = \ell(t_2) = (\epsilon, \epsilon) \,\}$$

In the analysis, we use $\mathcal{M}_2$, and since it is a possibly infinite language we need an algorithm to compute a finite representation of it. The function `M2` in Algorithm 2 returns a regular expression $\mathcal{R}_1 \in \mathbb{R}^\perp$ such that $\mathcal{L}(\mathcal{R}_1) = \mathcal{M}_2(\mathcal{R})$. In Algorithm 2, we compute the intersection of two regular expressions $\mathcal{R}_1, \mathcal{R}_2 \in \mathbb{R}^\mathcal{T}$ that does not include ϵ, and we denote it by $\mathcal{R}_1 \cap_{\not\epsilon} \mathcal{R}_2$. It can be computed as $\texttt{remove-eps}(\mathcal{R}_1) \cap \texttt{remove-eps}(\mathcal{R}_2)$, where $\texttt{remove-eps} : \mathbb{R}^\mathcal{T} \to \mathbb{R}^\perp$ removes ϵ from the language of input expressions. The procedure is depicted in Algorithm 3.

To ensure termination, we keep track of which stars have already been expanded with the parameter E. When a regular expression, in which the first construct is a star, is encountered for the second time, the function returns $\perp_r$. This guarantees that any star will be expanded exactly once. Observe that the closed

Algorithm 3: Remove ϵ from $\mathcal{L}(\mathcal{R})$

```
1  function remove-eps(R : R^T) → R^⊥
2      switch (regex-head(R), regex-tail(R)) do
3          case (ε, ε) ∨ (R1*‾, R2) do
4              return ⊥r
5          case (a, R1) do
6              return a · (refresh(R1))
7          case (R1 | R2, R3) do
8              return remove-eps(R1R3) | remove-eps(R2R3)
9          case (R1*, R2) do
10             return remove-eps(R1R1*‾R2) | remove-eps(R2)
```

stars and the parameter E serve different purposes: the first guarantees termination during the *concrete execution* to avoid infinite ϵ-matching loops; the second guarantees termination of the `M2-rec` function.

Theorem 4.1 (Correctness of `M2`)
Let $\mathcal{R} \in \mathbb{R}$.

$$\mathcal{L}(\texttt{M2}(\mathcal{R})) = \mathcal{M}_2(\mathcal{R})$$

Example 4.7 (`M2`$((a \mid a)^*)$)
Consider `M2`$((a \mid a)^*)$, that initially invokes `M2-rec`$((a \mid a)^*, \emptyset)$. First, $(a \mid a)(a \mid a)^{\overline{*}} \cap_{\not\epsilon} \epsilon =_{\mathcal{L}} \perp_r$ is returned; then, the recursive call `M2-rec`$(\epsilon, \emptyset)$ immediately terminates and returns $\perp_r$ as well. The most interesting recursive call is `M2-rec`$((a \mid a)(a \mid a)^{\overline{*}}, \{(a \mid a)^*\})$, where the first construct in the concatenation is an alternative. The function computes and returns the nonempty intersection $a(a \mid a)^{\overline{*}} \cap_{\not\epsilon} a(a \mid a)^{\overline{*}} =_{\mathcal{L}} a^+$. Next, the algorithm invokes `M2-rec`$(a(a \mid a)^{\overline{*}}, \{(a \mid a)^*\})$, which then calls `M2-rec`$(\texttt{refresh}((a \mid a)^{\overline{*}}), \{(a \mid a)^*\})$. Since `refresh`$((a \mid a)^{\overline{*}}) = (a \mid a)^*$ and $(a \mid a)^*$ is in E, the algorithm terminates at line 5. To summarize, `M2`$((a \mid a)^*)$ recognizes the language a^+, which is exactly $\mathcal{M}_2((a \mid a)^*)$.

Example 4.8 (Nested stars and ReDoS vulnerabilities)
Heuristics-based tools often classify as dangerous regular expressions that

have nested stars. In this example, we show how this pattern implies that the language of words that can be matched in at least two traces is non-empty. Consider the regular expression $(a^*)^*$ and the word aa. After matching the first character a, the matching engine reaches the state $(a^*(a^*)^*, a)$, as shown by the following partial trace:

$$((a^*)^*, aa) \xrightarrow{\circledast} (a^*(a^*)^{\overline{*}}, aa) \xrightarrow{\circledast} (aa^{\overline{*}}(a^*)^{\overline{*}}, aa) \xrightarrow{\circledcirc_a} (a^*(a^*)^*, a)$$

In this configuration, it is possible to match the subsequent character a by expanding either the left or the right star:

$$(a^*(a^*)^*, a) \xrightarrow{\circledast} (aa^{\overline{*}}(a^*)^*, a) \xrightarrow{\circledcirc_a} (a^*(a^*)^*, \epsilon)$$
$$(a^*(a^*)^*, a) \xrightarrow{\bigcirc} ((a^*)^*, a) \xrightarrow{\circledast} (a^*(a^*)^{\overline{*}}, a) \xrightarrow{\circledast} (aa^{\overline{*}}(a^*)^{\overline{*}}, a) \xrightarrow{\circledcirc_a} (a^*(a^*)^*, \epsilon)$$

This implies that the language of words that can be matched in at least two traces is non-empty. In general, nested stars can lead to this type of configuration in which words can be matched in two different concatenated stars. This implies that the regular expression might be dangerous, justifying the decision of heuristics-based tools to classify regular expressions with nested stars as vulnerable.
When analyzing $(a^*)^*$, after three recursive calls, `M2-rec` reaches the regular expression $a^*(a^*)^*$ and returns $a^* \cap_{\not\epsilon} (a^*)^* =_{\mathcal{L}} a^+$. This expression is then concatenated to the prefix that makes it possible to reach the configuration $a^*(a^*)^*$, namely a. Overall, the language of words that can be matched in at least two different traces is $a \cdot a^+$.

Intuitively, if there is no word that is matched in two different traces, there is no ambiguity, and the matching is linear in the length of the input words in the worst case. In Lemma 4.2, we formalize this intuition.

Lemma 4.2 (Linear matching with no ambiguity)
Let $\mathcal{R} \in \mathbb{R}^{\mathcal{T}}$.

$$\mathcal{M}_2(\mathcal{R}) = \emptyset \implies |[\![\mathcal{R}]\!](w)| = \mathcal{O}(|w|)$$

To understand how we take advantage of `M2`, consider a regular expression $\mathcal{R}^*$

such that $\texttt{M2}(\mathcal{R}^*) \not\equiv_{\mathcal{L}} \perp_r$. In this case, the set of words that are matched with at least two traces in $\mathcal{R}^*$ is not empty. Let $w \in \mathcal{L}(\texttt{M2}(\mathcal{R}^*))$. Since from $\mathcal{R}^*$ there are two traces to match w, then there are four traces to match w^2, eight for w^3, and in general 2^n for w^n. Furthermore, for all $n \geq 1$, $w^n \in \mathcal{L}(\texttt{M2}(\mathcal{R}^*))$. This implies that the words in $\texttt{M2}(\mathcal{R}^*)$ are possibly matched in an exponential number of traces. To have an exponential matching tree, all of them must be explored. Let $\mathcal{S} \in \mathbb{R}$, and consider the case in which w^n is matched with $\mathcal{R}^*\mathcal{S}$. By concatenating w^n with a suffix s that causes the match to fail, it is possible to force the procedure to exhaustively explore all traces, effectively resulting in an exponential matching tree. The language of suffixes that make the match fail is the language of words not accepted by $\mathcal{R}^*\mathcal{S}$, namely $\overline{\mathcal{R}^*\mathcal{S}}$. This is the key insight of our analysis, namely that $\texttt{M2}(\mathcal{R}^*) \cdot \overline{\mathcal{R}^*\mathcal{S}}$ accepts an overapproximation of the language of words dangerous for $\mathcal{R}^*\mathcal{S}$ that can cause exponential matching in $\mathcal{R}^*$.

With this intuition, we define the analysis $\mathcal{E} : \mathbb{R} \times \mathbb{R} \times \mathbb{R} \to \mathbb{R}^{\perp}$ such that $\mathcal{E}(\mathcal{R}, \mathcal{P}, \mathcal{S})$ recognizes an overapproximation of the set of words dangerous for the regular expression $\mathcal{P} \cdot \mathcal{R} \cdot \mathcal{S}$ that can cause exponential matching in $\mathcal{R}$.

Definition 4.3 (ReDoS analysis)
Let $\mathcal{R}, \mathcal{P}, \mathcal{S} \in \mathbb{R}$.

$$\mathcal{E} : \mathbb{R} \times \mathbb{R} \times \mathbb{R} \to \mathbb{R}^{\perp}$$

$$\mathcal{E}(\mathcal{R}, \mathcal{P}, \mathcal{S}) \triangleq \begin{cases} \perp_r & \text{if } \mathcal{R} = \epsilon \text{ or } \mathcal{R} = a \\ \mathcal{E}(\mathcal{R}_1, \mathcal{P}, \mathcal{S}) \mid \mathcal{E}(\mathcal{R}_2, \mathcal{P}, \mathcal{S}) & \text{if } \mathcal{R} = \mathcal{R}_1 \mid \mathcal{R}_2 \\ \mathcal{E}(\mathcal{R}_1, \mathcal{P}, \mathcal{R}_2 \cdot \mathcal{S}) \mid \mathcal{E}(\mathcal{R}_2, \mathcal{P} \cdot \mathcal{R}_1, \mathcal{S}) & \text{if } \mathcal{R} = \mathcal{R}_1 \cdot \mathcal{R}_2 \\ \mathcal{P} \cdot \mathcal{R}_1^* \cdot \texttt{M2}(\mathcal{R}_1^*) \cdot \overline{\mathcal{R}_1^* \cdot \mathcal{S}} \mid \mathcal{E}(\mathcal{R}_1, \mathcal{P} \cdot \mathcal{R}_1^*, \mathcal{R}_1^* \cdot \mathcal{S}) & \text{if } \mathcal{R} = \mathcal{R}_1^* \end{cases}$$

Initially, the analysis must be invoked as $\mathcal{E}(\mathcal{R}, \epsilon, \epsilon)$. It recursively explores $\mathcal{R}$, accumulating the prefixes and the suffixes of the portion that it is considering in $\mathcal{P}$ and $\mathcal{S}$. When $\mathcal{E}$ encounters a star, in addition to calling $\mathcal{E}$ recursively on the regular expression under the star, it also returns $\mathcal{P} \cdot \mathcal{R}_1^* \cdot \texttt{M2}(\mathcal{R}_1^*) \cdot \overline{\mathcal{R}_1^*\mathcal{S}}$. As discussed previously, $\texttt{M2}(\mathcal{R}_1^*)\overline{\mathcal{R}_1^*\mathcal{S}}$ recognizes an overapproximation of the language of words dangerous for $\mathcal{R}_1^*\mathcal{S}$ that can cause exponential matching in $\mathcal{R}_1^*$. The first construct $\mathcal{P} \cdot \mathcal{R}_1^*$ in the expression accepts the language of words that the analysis determined

to be a prefix of $\mathcal{R}_1^* \mathcal{S}$. Later in this section, we prove that the words in $\mathcal{E}(\mathcal{R}, \epsilon, \epsilon)$ are a sound overapproximation of the words that are dangerous for $\mathcal{R}$, and we also provide an example where the analysis loses precision.

We can perform an emptiness check on $\mathcal{E}(\mathcal{R}, \epsilon, \epsilon)$ to determine if there are dangerous words. If the language is empty, then $\mathcal{R}$ is not vulnerable; otherwise, we have a sound overapproximation of the words that can lead to ReDoS attacks.

Example 4.9 (ReDoS analysis)
Consider $\mathcal{E}((a \mid a)^*, \epsilon, \epsilon)$.

$$\begin{aligned}\mathcal{E}((a \mid a)^*, \epsilon, \epsilon) = \quad & (a \mid a)^* \cdot \mathtt{M2}((a \mid a)^*) \cdot \overline{(a \mid a)^*} \mid \mathcal{E}(a \mid a, (a \mid a)^*, (a \mid a)^*) \\ =_{\mathcal{L}} \ & (a \mid a)^* \cdot a^+ \cdot \overline{(a \mid a)^*} \mid \bot_r \\ =_{\mathcal{L}} \ & a^+ \cdot \overline{a^*}\end{aligned}$$

In this case, the analysis determined that $(a \mid a)^*$ is vulnerable to arbitrary large sequences of as that are followed by any nonempty word not composed of a's only. Observe that, effectively, $|[\![(a \mid a)^*]\!](a^n b)| = \Theta(2^n)$.

The following soundness theorem provides a strong guarantee that if the analysis of $\mathcal{R}$ returns an empty regular expression, then the size of any matching tree is at most polynomial in the length of the input word. More precisely, the matching is at most polynomial in the number of stars that syntactically appear in the regular expression.

Theorem 4.2 (Soundness of ReDoS analysis)
Let $\mathcal{R} \in \mathbb{R}$.

$$\mathcal{E}(\mathcal{R}, \epsilon, \epsilon) =_{\mathcal{L}} \bot_r \implies |[\![\mathcal{R}]\!](w)| = \mathcal{O}(|w|^{\mathtt{nstars}(\mathcal{R})})$$

In Appendix A we give the detailed proof of Thm. 4.2, and here we give the intuition. The idea is that, if $\mathcal{E}(\mathcal{R}, \epsilon, \epsilon)$ is empty, then we can bound the width of any matching tree for $\mathcal{R}$ to $\mathcal{O}(|w|^{\mathtt{nstars}(\mathcal{R})})$. Since the height of matching trees is linear in the length of input words (see Lemma 4.1), we can observe that the whole matching tree has size at most polynomial in $|w|^{\mathtt{nstars}(\mathcal{R})}$.

Example 4.10 (Loss of precision in ReDoS analysis)
Some patterns in regular expressions can cause a loss of precision in the analysis. Consider as example $\Sigma^* \mid (a \mid a)^*$ and observe how the matching procedure never explores the right (dangerous) branch of the outermost alternative. However, since the analysis does not consider the order in which the branches are explored (they are merged with the | constructor), it returns a non-empty attack language:

$$\mathcal{E}(\Sigma^* \mid (a \mid a)^*, \epsilon, \epsilon) = \mathcal{E}(\Sigma^*, \epsilon, \epsilon) \mid \mathcal{E}((a \mid a)^*, \epsilon, \epsilon) =_{\mathcal{L}} a^+\overline{a^*}$$

While our analysis can raise false positives, our experiments show that over 74,669 regular expressions taken from real-world use cases, this happens only in 49 instances. This shows that patterns that can make our analysis lose precision rarely occur in practice.

The fact that the analysis returns the language of dangerous words can be useful in different scenarios. For example, it is possible to use our algorithm in a matching engine that tries to match a word only if it is not in the attack language of the input regular expression. The analysis we put forward can also be integrated with a static analyzer for high-level programming languages: by paring our framework with a sound string analysis, it should be possible to prove the absence of ReDoS vulnerabilities in real-world applications. This is left as future work.

As discussed in Section 4.2.4, in this work we assume that the match is successful only if the entire word matches the regular expression (fullmatch semantics). Nevertheless, matching engines usually consider the match to be valid even if just a prefix of the word matches the expression (partial match semantics). To simulate this behaviour, we can simply append Σ^* at the end of the patterns. Observe that the complement of the universal language is $\bot_r$, so that if Σ^* is the only suffix of a dangerous star, the exponential behaviour cannot be triggered. As discussed in this section, this is because there exists no suffix that can make the match fail. The implication is that patterns that are dangerous in the fullmatch semantics, can be harmless in the partial match semantics. Since the latter is the one used in matching engines, our implementation (see Chapter 5) assumes it by default, but the translation between the two is trivial.

Remark 4.1 (Relation with abstract interpretation)
In this work, we did not express directly our analysis in the abstract interpretation framework [26] (discussed in Part III of this manuscript), as regular expressions are simpler entities than programs. However, we observe that there is a strong correlation between our framework and some underlying ideas in program analysis by abstract interpretation. In fact, similarly to abstract interpretation techniques for programs, our analysis proceeds by structural induction, accumulating semantic information on the reachable states of the matching engine. While in program analysis the *abstract domains* collect information about the values of the variables, in our analysis we collect information about the prefixes and suffixes of a regular expression with the $\mathcal{P}$ and $\mathcal{S}$ parameters. Similarly to abstract interpretation, we do not actually execute the matching (respectively, program), but we rather perform an "abstract execution", where we assume that any word (respectively, program memory) can be given as input. This is the fundamental reason why we lose precision, which again draws a parallel with program analysis by abstract interpretation. To summarize, even though we did not directly rely on the abstract interpretation framework, we leveraged the same underlying principles to put forward a sound analysis. We believe it would be possible to express our analysis directly in the abstract interpretation framework, even if we preferred a simpler characterization in terms of the size of matching trees.

4.5. Analysis extensions

In this section, we describe possible interesting extensions of our analysis, which we would like to explore in future work.

4.5.1. Backreferences

Backreferences are non-regular constructs, and they cannot be expressed using only regular patterns [81]. We believe that it is possible to automatically overapproximate regular expressions with backreferences in a sound way by substituting the backreferences with the capturing group that they refer to. For example, the Python

regular expression `(a)b(\1|a)*` could be substituted with `(a)b(a|a)*`, where `\1` has been replaced with `a`. This makes it possible to support regular expressions with backreferences without any modification to our analysis. The substitution overapproximates the language recognized by the expressions, and we believe it also preserves the existing ReDoS vulnerabilities. The fundamental observation is that backreferences at runtime are substituted with the string that is matched by the capturing group that they refer to. As a result, replacing them with the more expressive–and therefore potentially dangerous–capturing group with stars and unions does not eliminate any vulnerability. On the other hand, this technique could inject new vulnerabilities. Consider, for instance, the non-dangerous Python expression `(a)*b(\1)*` which recognizes the language $\{a^n b a^n \mid n \in \mathbb{N}\}$. If we replace the backreference `\1` with `a*`, we obtain `(a)*b(a*)*`, which, due to the nested star, presents a ReDoS vulnerability (see Example 4.8).

In order to prove the soundness of our substitution technique, our semantic framework must be extended to natively support backreferences. Then, we must prove that replacing the backreferences with the capturing group that they refer to does not discard any existing vulnerability. In future work, we would like to formally prove the soundness of this extension and implement it.

4.5.2. Lookaround assertions

Lookarounds are features that enable users to specify assertions on the characters that will be matched (or have been matched) by a pattern. In case the assertion is not respected by the input string, the match fails. For instance, the Python expression `a*(?!b)[a-z]` matches strings composed of an arbitrary number of `a`s, followed by any lower case letter that is *not* `b`. The expression `(?!b)` is a *negative lookahead*, namely a lookaround that asserts that the pattern `b` is *not* matched after `(?!b)`. Lookarounds can be *positive* when they assert that a pattern *is* matched, or *negative* if they assert that a patterns *is not* matched. Furthermore, they can be divided into *lookaheads* if the condition concerns what is matched after the assertion, or *lookbehinds* if they restrict the language of words that have been matched previously. Given that lookaround assertions can be encoded using only regular constructs [82, 83, 84], it is tempting to believe that we can straightforwardly transform regular expressions with lookarounds into automata, and subsequently

revert these automata back into expressions without lookarounds. This would make it possible to support lookarounds without any modification to our analysis. Nevertheless, this approach has a major pitfall: the known conversion methods from regular expressions to automata (see Section 3.3) *are not guaranteed to preserve ReDoS vulnerabilities*. This implies that the aforementioned technique could possibly lead to both false positives and false negatives during the analysis. In Section 4.6.1 we discuss how this approach is a common pitfall for static analysis ReDoS detection techniques.

A simple solution is to run our analysis on the input expression ignoring the lookarounds, and then run separate analyses for each assertion. If none of them presents a vulnerability, then the overall expression is safe. For instance, in order to analyze `a*(?!(b|b)*)[a-z]`, we should consider separately the expression without lookarounds (namely `a*[a-z]`), and then `(b|b)*`. The analysis would classify the original expression as dangerous, as `(b|b)*` is vulnerable. This method has the disadvantage of not being able to capture an overapproximation of the attack language, as each assertion is analyzed individually. Nevertheless, the analysis is sound, as it classifies as dangerous all the expressions that present a vulnerability.

A better solution would be to extend our semantic framework to natively support lookarounds. This would make it possible to formally reason on the assertions on a semantic level, and allow us to extract an overapproximation of the attack language. An example of formal semantics for lookaround assertions is given in [84].

4.5.3. Superlinear matching analysis

In this work, we focused our attention on exponential ReDoS vulnerabilities. Nevertheless, superlinear matching could be dangerous if the exponent of the polynomial is high. Consider, for instance, what happens if the string $a^n b$ is matched against the regular expression a^*a^*. The word can be matched in n different ways: it is enough to choose an index i, for $0 \leq i \leq n$, at which a^n is split. Then, the first star matches a^i, and the second matches a^{n-i}. Every match costs up to n steps, all of which will be expanded due to the fact that the engine fails to accept the suffix b. As a result, the matching costs quadratic time.

While in the exponential analysis it is enough to consider the language of words that can be matched in two traces in a single star, this reasoning is not sufficient

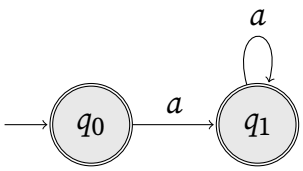


FIGURE 4.3. Glushkov's automaton for $(a^*)^*$

for superlinear matching. In fact, superlinear behavior arises when the ambiguity is due to stars that recognize the same language and are concatenated. Since reasoning on each individual star is not sufficient, superlinear matching is more challenging to detect. A superlinear analysis should explore a regular expression by induction, keeping track of the prefixes and suffixes for each individual constructor. Then, once a star is detected, the analysis should intersect the language of words recognized by the star with the language of the suffixes. If the intersection is not empty and contains words of arbitrary length, this implies that there exists a suffix star that accepts the same language. Therefore, superlinear matching can potentially occur.

Observe that being able to determine the exponent of the polynomial is fundamental: quadratic and cubic matching is, in most real-world cases, not considered dangerous. Since linear and quadratic matching behave similarly in terms of number of matching steps, dynamic ReDoS detectors such as RESCUE [73] usually do not report those vulnerabilities. Once the matching has an exponent of four or more, then the vulnerability is considered dangerous. While the detection of superlinear matching can be implemented with the aforementioned technique, it is still not clear how to determine the exponent of the polynomial.

4.6. Related work

In this section, we discuss related work. In particular, we describe existing ReDoS detection techniques, ReDoS mitigation frameworks, and the relation between our semantics and regular expression derivatives [63, 99].

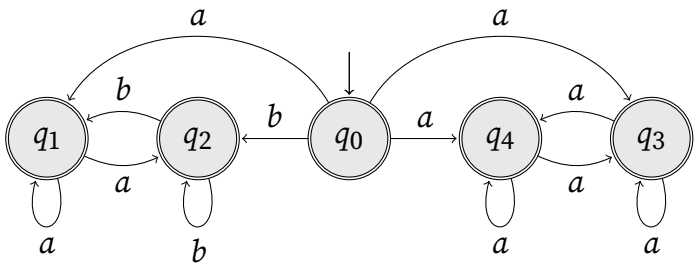


FIGURE 4.4. Glushkov's automaton for $\Sigma^* \mid (a \mid a)^*$ over $\Sigma = \{a, b\}$

4.6.1. Semantics-based static ReDoS detection

Wüstholz et al. [72] put forward an analysis based on automata to detect ReDoS vulnerabilities, and they implement it in the REXPLOITER tool. The authors classify an automaton as *vulnerable* when it presents a state from which a word can be matched by following two different paths, and then come back to the same state (see [72, Thm. 1]). Their approach is the closest to ours, since they can as well extract the language of dangerous words. However, the analysis is not sound nor complete, because transforming a regular expression into an automaton can introduce or remove vulnerabilities. For example, by applying Glushkov's construction (see Section 3.2) to the vulnerable regular expression $(a^*)^*$ we obtain the non-vulnerable automaton (with respect to the definition of vulnerability given in [72]) represented in Figure 4.3. Since they do not define an algorithm to transform regular expressions into automata that preserves vulnerabilities, the analysis can report both false positives and false negatives, and our experiments described in Chapter 5 confirmed this. Observe that any approach based on traditional automata cannot be complete. Since finite automata lack a mechanism for prioritizing transitions between states, any attempt to precisely capture all vulnerabilities with respect to the original expression is hopeless. Consider for instance the non-vulnerable regular expression $\Sigma^* \mid (a \mid a)^*$ over the alphabet $\Sigma = \{a, b\}$ (see Example 4.10). If we apply Glushkov's construction to it, we obtain the automaton in Figure 4.4. While the regular expression is non-vulnerable due to the fact that the dangerous subexpression $(a \mid a)^*$ is never expanded, the automaton is vulnerable with respect to the definition of vulnerability given in [72].

The RXXR2 tool is a static analyzer for exponential ReDoS vulnerabilities that infers exploit strings [70]. It is the successor of RXXR [69], that turned out to be

unsound. Introducing a novel approach based on NFAs with prioritized transitions, RXXR2 infers strings that can be *pumped* and lead to exponential matching. While the algorithm is sound and complete with respect to automata, transforming regular expressions to automata can introduce or remove vulnerabilities. Similarly to REXPLOITER, they assume that the input expression has been converted into an automaton following one of the standard constructions, so that the analysis is actually neither sound nor complete.

The framework of *prioritized NFAs* (pNFAs) [89, 105] has been leveraged by Weideman et al. [71] to build the RSA (**R**egex**S**tatic**A**nalysis) static analyzer. The authors introduce an algorithm to translate regular expressions into automata that preserves the ReDoS vulnearbilities. The automata are analyzed with the framework described in [106] to determine the *degree of ambiguity* [107], which allows inferring whether there are ReDoS vulnerabilities or not. The *full* mode performs a sound and complete analysis, while the *simple* mode is only sound, but it usually runs faster. We observe that while the analysis is complete, it is strictly less expressive than ours. In fact, their framework cannot be used to extract the attack language for a regular expression, but only a finite number of exploit strings. For this reason, the two approaches are suitable for different uses: tools that need the specification of dangerous words, such as static analyzers, cannot rely on RSA to extract it. Furthermore, our algorithm performs a single *emptyness check* of the attack language, while their analysis performs a *universality check* for each state of the automaton, resulting in a strictly higher complexity. Our experiments (see Chapter 5) confirm that our analysis has a substantial performance advantage over the one proposed in [71]. While our ReDoS detector and RSA are the only tools based on sound techniques, the latter still produces false positives and false negatives in practice, possibly due to bugs in the implementation (see Chapter 5).

4.6.2. Dynamic ReDoS detection

A radically different approach to ReDoS detection is dynamic analysis. The RESCUE tool [73] leverages a genetic algorithm to efficiently generate potentially dangerous words, that are then matched by the Java matching engine to determine if they are truly dangerous. For this reason, the tool cannot report false positives. On the other hand, there is no guarantee about the absence of false negatives. The

gray-box approach makes it easy to support a wide variety of advanced features, but it has the disadvantage of being several orders of magnitude slower than static analyzers. The analysis is not deterministic, and due to its dynamic nature it is not expressive enough to compute the attack language.

Generic fuzzers, such as SLOWFUZZ [103], can be configured to detect ReDoS vulnerabilities. Even if they can be effective to report true vulnerabilities, a gray-box approach that has (partial) knowledge about the underlying matching engine is more effective [73].

4.6.3. Heuristics-based static ReDoS detection

Heuristics-based static analyzers try to report vulnerabilities by matching potentially dangerous patterns against the constructors of a regular expression. For instance, SAFE-REGEX [75] checks that expressions do not present nested stars. It is easy to craft an example for which this rule raises a false positive: the regular expression $(a^*)^*\Sigma^*$ has two nested stars, but since there is no suffix that can make the matching fail (Σ^* accepts the universal language), the expression is not dangerous. Nevertheless, SAFE-REGEX reports that the expression is dangerous, effectively raising a false positive. In our experiments described in Chapter 5, we also found that both SAFE-REGEX and REGEXPLOIT raise a false negative when analyzing the Python regular expression `<project(.|\s)*?>`, as they do not detect the exponential vulnerability. The exponential behaviour can be triggered by using as exploit string `<project` followed by a sequence of space characters, since spaces can be matched by both branches of the alternative `(.|\s)`. Heuristics-based analyzers do not have semantic information about the attack language, and they do not perform dynamic testing either. In our experiments, we observed that these tools report a high number of false positives and false negatives. The heuristics employed by SAFE-REGEX [75], REGEXPLOIT [76] and REDOS-DETECTOR [77] are not formalized in any work, and they can potentially change in the future. Not having semantic information also implies that it is impossible for this class of detectors to differentiate the type of the vulnerabilities reported, namely if the matching is exponential or superlinear.

4.6.4. ReDoS mitigation

Recently, many techniques have been proposed to mitigate ReDoS attacks. Cody-Kenny et al. [108] use genetic programming to substitute vulnerable regular expressions with safe ones. Li et al. [109] and Pan et al. [110] put forward techniques for automatic expression repair based on examples. In [111] the authors introduce a matching algorithm that leverages selective memoization to mitigate ReDoS attacks while supporting advanced regular expression features. The matching algorithm proposed in [84] supports lookaround assertions, while keeping the matching linear in the length of the words. Sophisticated techniques based on GPU matching [112, 113] and state-merging algorithms [114] have also been proposed to speedup the matching.

4.6.5. Regular expression derivatives

Derivatives-based matching [63, 99, 61] is a technique to perform regular expression matching. It relies on the fundamental concept of *derivative of a regular expression* (see Section 3.2). In general, given a symbol a, the derivative of a regular expression $\mathcal{R}$ with respect to a is an expression that recognizes only those suffixes of strings with a leading a accepted by $\mathcal{R}$. Brzozowski's derivatives [63] are related to DFAs, while Antimirov's partial derivatives [99] are related to NFAs, and both can be leveraged to perform regular expression matching. Matching engines in widely used programming languages do not use derivatives-based matching, as they rely on backtracking algorithms [79, 80, 89].

There are some similarities between our tree semantics and Brzozowski's derivatives. The connection lies in the fact that when we match the first character from the state $(\mathcal{R}, aw)$, the regular expressions that we find in the states after matching a recognize the same language accepted by the derivative of $\mathcal{R}$ with respect to the character a. Nevertheless, there are substantial differences between the two approaches. In fact, our semantics is designed to capture the exact states explored by the matching engine, and in which order they appear. For instance, we can observe that after matching the first a starting from $((a \mid a)^*, ab)$, we explore the state $((a \mid a)^*, b)$ exactly twice. This would not be possible by using derivatives, as they do not enjoy a notion of order over the expanded expressions. Furthermore, to mimic the behaviour of matching engines we added the *closed star* constructor,

which is not needed in derivatives. Since regular expression derivatives cannot precisely capture the state of the matcher, they are not suitable to formally describe and reason about ReDoS vulnerabilities.

4.7. Conclusion

In this chapter, we defined a tree semantics for regular expression matching, which we leveraged to design a sound static analysis that detects ReDoS vulnerabilities. To the best of our knowledge, our ReDoS detection framework is the first one that operates directly on regular expressions without having to resort to automata. This allowed us to easily reason about the concrete behaviour of complex matching engines. Our approach is *semantic*, namely rooted in the formal definition of the behaviour of the matching procedure, which draws a parallel with program analysis techniques such as abstract interpretation.

To assess the usefulness of our analysis, we implemented it in a tool called RAT, and we compared it to seven other ReDoS detectors. In Chapter 5 we describe our implementation and our experimental results in detail. Our experiments show that our tool is the only detector that is sound also in practice, as all other detectors report false negatives. RAT does not raise any false negatives, which matches our theoretical results and gives empirical evidence of the fact that our analysis is effectively sound.

In future work, we would like to extend our analysis to support advanced features such as backreferences and lookarounds. We believe that it is possible to automatically overapproximate those features with regular constructs in a sound way. We would also like to use the matching semantics to design a detector for superlinear ReDoS vulnerabilities. Similarly to the exponential case, we expect that an approach based on regular expressions can lead to an efficient and sound analysis also for superlinear vulnerabilities. Another interesting extension of this analysis would be to integrate our framework in a static analyzer for high-level languages such as Python. We believe that by pairing our detection technique with a string analysis, it is possible to prove the absence of ReDoS vulnerabilities in real-world applications. During the analysis, when a regular expression matching is found, the analyzer would intersect the language of dangerous words with the

possible values that the matched string can have. If the intersection is empty, this proves that the match is safe.

Chapter 5

ReDoS Analysis Experimental Evaluation

To assess the usefulness of the ReDoS analysis described in Chapter 4, we have implemented it in a tool called RAT. We tested it on a dataset of 74,669 regular expressions, and we observed that in 99.78% of the instances the analysis terminates in less than one second. We compared RAT to seven other ReDoS detectors, and we found that our tool is faster, often by orders of magnitude, than most other tools. While raising a low number of false positives, RAT is the only ReDoS detector that does not report false negatives. Our approach based on regular expressions not only eliminates the complexities related to using automata, but also opens the possibility to easily introduce optimizations.

In this chapter, we present the results obtained in our experimental evaluation. First, in Section 5.1 we describe our experimental setup. Then, in Section 5.2 and Section 5.3 we respectively present the performance and precision results, which we further comment in Section 5.4.

5.1. Experimental setup

To assess the usefulness of the analysis we put forward, we implemented it in the RAT [40] tool (**R**eDoS **A**bstract **T**ester, which is publicly available on Github) in less than 5000 lines of OCaml code, and we compared it to four other detectors.

TABLE 5.1. Attributes of the ReDoS detectors

	Type	Sound	Complete	Language	Deterministic
RAT	*static, semantic*	✓	✗	✓	✓
RESCUE [73]	*dynamic*	✗	✓	✗	✗
REXPLOITER [72]	*static, semantic*	✗	✗	✓	✓
RSA [71]	*static, semantic*	✓	✗	✗	✓
RSA-FULL [71]	*static, semantic*	✓	✓	✗	✓
RXXR2 [70]	*static, semantic*	✗	✗	✗	✓
SAFE-REGEX [75]	*static, heuristic*	✗	✗	✗	✓
REGEXPLOIT [76]	*static, heuristic*	✗	✗	✗	✓
REDOS-DETECTOR [77]	*static, heuristic*	✗	✗	✗	✓

In Appendix B we present significant implementation details of our tool. In our experiments, we wanted to evaluate how RAT behaves in terms of precision and performance compared to seven other ReDoS detectors. We ran our experiments on a server with 128GB of RAM, with 48 Intel Xeon CPUs E5-2650 v4 @ 2.20GHz and Ubuntu 18.04.5 LTS. We considered the dataset used in [73], composed of: (1) 2,992 patterns from the Regexlib platform [102]; (2) 12,499 patterns from the Snort platform [115]; (3) 13,597 patterns extracted from 3,898 Python projects on Github in [67]. To them, we added 63,352 regular expressions extracted from modules in the PYPI package manager [116] by Davis et al. [117]. From the dataset, we removed the expressions that were not properly sanitized (e.g., that contained non-printable characters) and we removed duplicates, obtaining 74,669 regular expressions. To the best of our knowledge, it is the first time that such a large dataset of regular expressions taken from real-world programs is used to compare the precision and performance of ReDoS-detection tools.

In what follows, we say that a detector is *sound* if it identifies as vulnerable all the truly vulnerable regular expressions, and we say that it is *complete* if all the expressions it identifies as vulnerable are truly vulnerable. Sound detectors forbid *false negatives*, while complete detectors forbid *false positives*. The tools we compared RAT to are RESCUE [73], REXPLOITER [72], RSA [71], RXXR2 [70], SAFE-REGEX [75], REGEXPLOIT [76] and REDOS-DETECTOR [77]. In particular, RSA allows the user to improve the precision of the analysis (at the cost of sacrificing some performance) with the "full" mode, that makes it the only sound and complete tool. In our experiments, we consider the regular and the full modes of RSA as

two different analyzers. The only dynamic detector we compare to is RESCUE that, due to its nature, never raises false positives. On the other hand, since it relies on a genetic algorithm that generates the input strings with random mutations, the analysis is not deterministic. The detectors SAFE-REGEX, REGEXPLOIT and REDOS-DETECTOR are heuristics-based, and they do not offer any guarantees about the soundness nor the completeness of the analysis. In Table 5.1 we summarize the characteristics of the tools. While attributes reported in Table 5.1 summarize the *expected* behaviour, we found that in practice some detectors do not match the underlying theoretical results. For instance, while RSA-FULL should be sound and complete, we found that it reports both false positives and false negatives. This is probably due to bugs in the implementation. If a detector can extract the language of dangerous words (as opposed to a single exploit string) we mark the **Language** column with ✓. Static detectors are divided into semantics-based and heuristics-based tools.

5.2. Precision comparison

We take advantage of the evaluation technique used in [73], which, to the best of our knowledge, is the only article that compares the precision of different ReDoS detectors. We analyze each regular expression with the detectors setting an individual timeout of 30 seconds, and then we compare the results. If any tool can craft an exploit string of length lesser or equal to 128 characters that makes the Java 8 matching engine perform more than 10^{10} matching steps, we consider the expression to be vulnerable. During our tests, we observed that for the specific matching engine we consider, for strings of length at most 128 characters, 10^{10} matching steps are a sound threshold to clearly distinguish between exponential and non-exponential matching. We cross-reference the results of eight different tools (some of which are, at least theoretically, sound) by concretely testing exploit strings on a real-world matching engine, so that we infer with high confidence the number of false positives and false negatives. Nevertheless, since we include RESCUE in the comparison, which is a nondeterministic detector, these numbers might vary slightly in different runs. We classified as vulnerable 316 regular expressions.

In Table 5.2, we report the results of the comparison. The columns correspond

TABLE 5.2. ReDoS detectors precision evaluation results

	OK	FP	FN	OOT	AC	SKIP	TIME
RAT	67,052	49	0	178(21)	0(0)	7,390(13)	1:57:20
RXXR2 [70]	60,794	93	7	10(2)	0(0)	13,765(23)	0:09:29
RESCUE [73]	33,531	0	40	32,208(43)	0(0)	8,890(34)	325:00:26
RSA [71]	57,269	193	1	789(47)	240(35)	16,177(42)	18:48:02
RSA-FULL [71]	54,857	134	1	3,138(55)	400(43)	16,139(42)	38:11:07
REXPLOITER [72]	53,931	28	180	328(1)	0(0)	20,202(104)	9:12:34
SAFE-REGEX [75]	61,272	13,376	21	0(0)	0(0)	0(0)	0:15:40
REGEXPLOIT [76]	74,050	56	140	2(0)	0(0)	421(14)	0:03:41
REDOS-DETECTOR [77]	45,694	14,218	6	2(1)	0(0)	14,749(92)	0:52:27

to: number of correctly classified regular expressions (**OK**); false positives (**FP**); false negatives (**FN**); out of time (**OOT**); analyzer crashes (**AC**); skipped (**SKIP**) (i.e., not parsed); total runtime displayed as HH:MM:SS (**TIME**). For out of time events, analyzer crashes, and skipped regular expressions, we report in parentheses how many expressions in the total number are vulnerable.

Compared to other static analyzers, RAT reports a relatively low number of false positives: 49 over the 67,074 regular expressions that it parses. The only static analyzer that reports fewer false positives than RAT is REXPLOITER, that on the other hand reports respectively 180 false negatives. Furthermore, REXPLOITER skips 20,202 regular expressions. Interestingly, we observed that in practice RAT *is the only detector that does not report false negatives*. This matches our theoretical results, and it gives empirical evidence that our framework performs a sound analysis.

If we do not consider heuristics-based tools, RAT is the detector that parses the highest number of expressions: even more than RESCUE, which indeed supports advanced features. This is due to the fact that RESCUE does not support some regular patterns such as *named capturing groups* with the syntax `(?P<name>pattern)`, that indeed RAT can analyze. Heuristics-based detectors can analyze a higher number of expressions: REGEXPLOIT and SAFE-REGEX skip respectively only 421 and 0 regular expressions. Since these tools do not offer guarantees about the soundness or the completeness of the analysis, they can analyze a wide variety of constructs by simply ignoring them. On the other hand, we observe that SAFE-REGEX parses and analyzes regular expressions that, to the best of our knowledge, are not accepted

by any matching engine. Examples include regular expressions with unclosed parentheses, for instance `(a`. The high number of false positives reported by SAFE-REGEX and REDOS-DETECTOR makes it difficult to use them in practice. In fact, they raise respectively 13,376 and 14,218 false alarms.

5.3. Performance comparison

In case a detector runs out of time for a few regular expressions, the total runtime in Table 5.2 grows sharply, not representing precisely the average performance of the tool. For this reason, we use *survival plots* to compare more faithfully the performance of the detectors. On such a plot, the y-axis represents the time in milliseconds, and the x-axis is the number of expressions such that each one can be analyzed under the specified time, while the remaining regular expressions either take longer to analyze or cannot be analyzed by the corresponding detector. No plot for x-axis and detector d means that for 74,669–x expressions d did not successfully complete the analysis (i.e., it either ran out of time or it had a parse/runtime error). The plot highlights the relative performance of each tool and how many regular expressions can be individually analyzed under a time threshold. The survival plot of our experiments is depicted in Figure 5.1.

Our experiments showed that RAT is able to analyze 66,926 regular expressions over the 67,074 that it parses in less than one second each ($\sim$ 99.78%). As expected, RESCUE is, due to its dynamic nature, significantly slower than static analyzers. After it, we find the cluster composed of RSA, RSA-FULL and REXPLOITER. Our detector is on average one to two orders of magnitude faster than them for corresponding points on abscissa x. Even though the total runtime to analyze the whole dataset for REDOS-DETECTOR is lower than RAT's, the plot shows how our tool performs significantly better on average. The same holds for SAFE-REGEX: in 82,8% of the cases RAT is faster. The REGEXPLOIT tool performs better than our analyzer, at the cost of raising 140 false negatives, meaning that it does not detect more than one third of the vulnerabilities. While RXXR2 is generally faster than RAT, we remark that RAT is performing a strictly more expressive analysis by returning the language of dangerous words. Furthermore, according to Table 5.2, RXXR2 is not performing a sound analysis either. We also remark that RAT analyzes

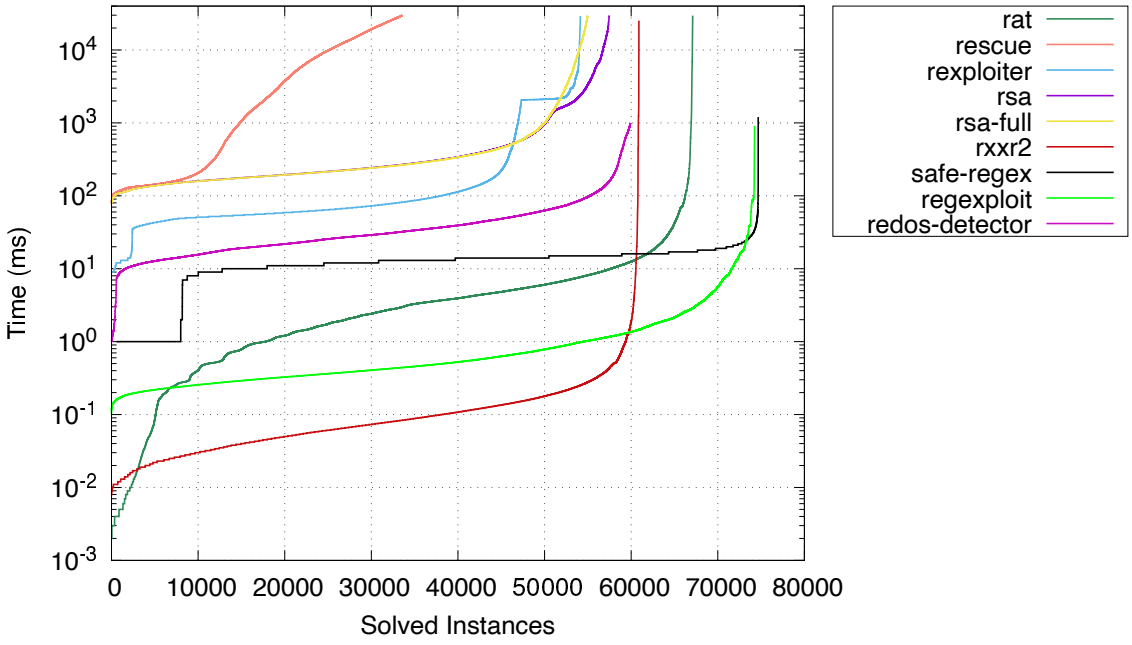


FIGURE 5.1. Survival plot with a logarithmic y axis and linear x axis

6,375 more expressions than RXXR2.

5.4. Discussion

We observed that in practice RAT is one to two orders of magnitude faster than most detectors, raises a relatively low number of false positives, and it is the only analyzer that does not report false negatives. The approach based on semantic trees significantly improved the design of the analysis and the ease of reasoning about ReDoS vulnerabilities. It also allowed us to ignore the complexities related to transforming regular expressions into automata, that for some tools are sources of unsoundness and incompleteness. To the best of our knowledge, our analysis for ReDoS vulnerabilities is the first that operates directly on regular expressions without having to resort to automata. Regular expressions also make it easy to implement many performance optimizations. We integrated in RAT three major performance improvements, which we further discuss in Appendix B.

Character classes representation. Character classes are features commonly used

by programmers. For example, `\d` is a shortcut for $0 \mid 1 \mid \ldots \mid 9$. We extend the regular expressions to recognize *sets of characters* instead of simple characters. With a slight adjustment to our implementation, expressions containing character classes considerably decreased their size. For example, $0 \mid \ldots \mid 9$ has 19 constructs, while $\{0, \ldots, 9\}$ is a regular expressions with a single character set construct.

Symbolic operations. In our analysis, we perform a large number of intersection and complement operations. Instead of running the algorithm to compute them, we use *extended regular expressions* (see Section 3.2) in order to support *symbolic intersection* and *symbolic complement*. When a complement or an intersection must be computed, we simply add its symbolic representation to the result, which is a constant time operation.

Emptiness check. The last step in the analysis is to check if the computed attack language is empty. We decided to take advantage of the algorithm based on *derivatives* put forward in [62], which, as the results of our experiments confirm, efficiently performs the emptiness check. The framework described in [62] uses *extended regular expressions* with symbolic intersection and symbolic complement, so that it can be effortlessly integrated into our implementation.

We conducted an analysis to determine the number of regular expressions that produce false positives in both RAT and other tools. Our investigation found that RXXR2 and RSA share respectively 21 and 26 false positives with RAT. This overlap is significant and can be attributed to similarities in the approaches used by these detectors. Typically, automata-based tools leverage analysis techniques to detect nondeterministic transitions in the loops of the automata. Our algorithm `M2` performs a similar analysis on the stars of regular expressions, as it detects the language of words that can be matched in two different traces. As stars in regular expressions are often transformed into loops in automata, we can account for the shared false positives between RAT and automata-based tools.

Upon examining the false positives reported by other tools, we found no correlation with RAT. In the case of SAFE-REGEX and REDOS-DETECTOR, the number of false alarms generated was too high to draw meaningful conclusions. In the case

of REGEXPLOIT, the overlap is limited to 10 false positives, while RESCUE cannot report false alarms. Although REXPLOITER employs an automata-based algorithm, only two false positives were shared between the tool and RAT. This finding highlights that the translation algorithm used by REXPLOITER fails to preserve the structure, and therefore the vulnerabilities, of regular expressions.

In our experimental evaluation, we did not find any recurring pattern in the false positives raised by RAT. By considering a large set of regular expressions that result in false positives, we might build a database of rules to improve the precision of the analysis in specific cases. Nonetheless, the soundness guarantee offered by our theoretical framework does not trivially hold if we add human-crafted ad-hoc rules to our analyzer. As a result, any rule that is used must be proved to preserve the soundness of the analysis.

5.5. Conclusion

We implemented our ReDoS detection framework in the RAT tool, and to assess the effectiveness of our technique, we compared it to seven other detectors. We found RAT to be on average one to two orders of magnitude faster than most tools, while giving strong guarantees about the soundness of the analysis. While raising a relatively low number of false positives, RAT is the only ReDoS detector that did not report false negatives. Our implementation is open source and available on GitHub [40].

In future work, we would like to integrate into RAT the analysis extensions discussed in Chapter 4 (see Section 4.5). For instance, it would be interesting to add support for detecting superlinear but not exponential ReDoS vulnerabilities. Benchmarking such an analysis requires determining the number of matching steps to differentiate between superlinear and linear matching. This can be challenging, especially when the exponent of the polynomial is low, as quadratic and linear matching behave similarly. Another interesting extension of our work would be to enhance RAT with support for backreferences and lookarounds. We believe that incorporating support for these advanced features into RAT will enable us to analyze the entire dataset of 74,669 regular expressions. This would make RAT the only tool capable of supporting such a large number of expressions while

maintaining soundness.

Part III

Verification of Security Properties for Programs

Chapter 6

Static Analysis by Abstract Interpretation

In this chapter, we study the principles of static analysis by *abstract interpretation* [26], which is technique co-invented in the late 70's by Patrick and Radhia Cousot. We start by introducing in Section 6.1 the syntax of a toy language called WHILE, which features classic iterative constructs such as while loops and assignments. Then, in Section 6.2, we give a mathematical formalization of the behaviour of the language, namely its *semantics*. By reasoning on the semantics of WHILE, in Section 6.3, we introduce different classes of *program properties*. In Section 6.4, we finally introduce the abstract interpretation framework, which makes it possible to prove properties of certain programs.

6.1. Syntax

In Figure 6.1 we report the grammar of the WHILE language. The set $\mathbb{V}$ is the finite set of program variables, and programs are defined as statements. WHILE has classic imperative features such as assignments `x = A` and conditionals `if (B)` $\mathtt{S}_t$ `else` $\mathtt{S}_e$. Arithmetic expressions operate on integers, and, as we will further discuss in Section 6.2, can incur runtime errors due to divisions by zero. Boolean expressions evaluate to true (`tt`) or false (`ff`), and, since they can have arithmetic expressions as subexpressions, can result in runtime errors as well.

$$
\begin{aligned}
&\texttt{P} := \texttt{S} && \text{(Programs)}\\
&\texttt{S} := \texttt{skip} && \text{(Statements)}\\
&\quad \mid \texttt{x = A}\\
&\quad \mid \texttt{S; S}\\
&\quad \mid \texttt{if (B) S else S}\\
&\quad \mid \texttt{while (B) S}\\
&\texttt{A} := \texttt{n} \in \mathbb{Z} && \text{(Arithmetic Expressions)}\\
&\quad \mid \texttt{x} \in \mathbb{V}\\
&\quad \mid \texttt{A} \diamond \texttt{A} \ \ (\diamond \in \{\texttt{+}, \texttt{-}, \texttt{*}, \texttt{/}\})\\
&\texttt{B} := \texttt{tt} && \text{(Boolean Expressions)}\\
&\quad \mid \texttt{ff}\\
&\quad \mid \neg\texttt{B}\\
&\quad \mid \texttt{B} \diamond \texttt{B} \ \ (\diamond \in \{\texttt{\&\&}, \texttt{||}\})\\
&\quad \mid \texttt{A} \diamond \texttt{A} \ \ (\diamond \in \{\texttt{<}, \texttt{<=}, \texttt{>}, \texttt{>=}, \texttt{==}, \texttt{!=}\})
\end{aligned}
$$

FIGURE 6.1. Syntax of the WHILE language

Example 6.1 (WHILE program computing the factorial)
The following program computes the factorial of `n`. The variable `factorial` holds the result of the computation.

```
1  if (n <= 1) {
2    factorial = n;
3  } else {
4    factorial = 1;
5    i = 2;
6    while (i <= n) {
7      factorial = factorial * i;
8      i = i + 1;
9    }
10 }
```

Remark 6.1 (If statements with no else branch)
If statements that consist only of the then branch are not permitted by our

grammar. However, they can be defined as *syntactic sugar* using regular if statements as follows:

$$\texttt{if (B) } \mathtt{S}_t \triangleq \texttt{if (B) } \mathtt{S}_t \texttt{ else skip}$$

We sometimes use program labels to explicitly refer to program points in statements. We assume that there is a finite set of labels $\mathbb{L}$, and that statements are annotated with these unique labels. When necessary, we explicitly report the labels. Since if and while statements need to differentiate the last label in the statement from the last label in the substatements, we use the following syntax to clearly separate the two:

$$\ell_1\texttt{if (B) }\ell_2\mathtt{S}_t\ell_3\texttt{ else }\ell_4\mathtt{S}_e\ell_5\texttt{fi}\ell_6$$
$$\ell_1\texttt{while }\ell_2\texttt{(B) }\ell_3\mathtt{S}_b\ell_4\texttt{od}\ell_5$$

Since we only rarely need to explicitly annotate the programs with the labels, for the sake of compactness, most of the times we do not report them.

6.2. Semantics

The *semantics* of a programming language is a precise mathematical description of the language's behaviour. To formally reason about the behaviour of programs, and ultimately *prove* properties about those programs, we need a precise formal framework to reason about them. Note that the semantics of programming languages, such as C and JavaScript, are typically formalized in English within specification documents [118, 119]. These documents often leave room for interpretation, leading to potential misunderstandings. Formal methods take a different approach to the problem, formalizing the semantics of programs as mathematical entities that do not leave space for human interpretation. In this section, we formally define the semantics of the WHILE language. We first study the *reachability semantics*, which captures, for each program point, the set of reachable states. Then, we study a more informative semantics known as the *trace semantics*, which we use to define different classes of program properties.

6.2.1. Expressions semantics

Program memories are the fundamental building blocks for our semantics. They are defined as functions from variables to integers, namely $m \in \mathbb{M} \triangleq \mathbb{V} \to \mathbb{Z}$. We denote the memory entries as $\mathtt{x} \mapsto \mathtt{n}$. We define the memory update $m[\mathtt{x} \leftarrow n]$ as follows:

$$m[\mathtt{x} \leftarrow n] \triangleq \{\, \mathtt{x} \mapsto n \,\} \cup \{\, \mathtt{y} \mapsto n' \mid (\mathtt{y} \mapsto n' \in m) \wedge (\mathtt{y} \neq \mathtt{x}) \,\} \tag{6.1}$$

The value $\lightning$ represents a runtime error, and $\mathbb{Z}_{\lightning} \triangleq \mathbb{Z} \cup \{\, \lightning \,\}$. The *arithmetic evaluation* $\mathcal{A}[\![\mathtt{A}]\!] : \mathbb{M} \to \mathbb{Z}_{\lightning}$ definition is straightforward: it results in an error if there is a division by zero. In our semantics, we consider only runtime errors that arise from divisions by zero, i.e., there is no notion of overflow (we use perfect mathematical arithmetic).

$$\mathcal{A}[\![\mathtt{A}]\!] : \mathbb{M} \to \mathbb{Z}_{\lightning}$$

$$\mathcal{A}[\![\mathtt{n}]\!]m \triangleq \mathtt{n} \tag{6.2}$$

$$\mathcal{A}[\![\mathtt{x}]\!]m \triangleq m(\mathtt{x}) \tag{6.3}$$

$$\mathcal{A}[\![\mathtt{A}_1 \diamond \mathtt{A}_2]\!]m \triangleq \begin{cases} \lightning & \text{if } \diamond = / \text{ and } \mathcal{A}[\![\mathtt{A}_2]\!]m = 0 \\ \lightning & \text{if } \mathcal{A}[\![\mathtt{A}_1]\!]m = \lightning \text{ or } \mathcal{A}[\![\mathtt{A}_2]\!]m = \lightning \\ \mathcal{A}[\![\mathtt{A}_1]\!]m \diamond \mathcal{A}[\![\mathtt{A}_2]\!]m & \text{otherwise} \end{cases} \tag{6.4}$$

The set $\mathbb{B}$ is $\{\, \mathtt{tt}, \mathtt{ff} \,\}$, and $\mathbb{B}_{\lightning} \triangleq \mathbb{B} \cup \{\, \lightning \,\}$. The *boolean evaluation* $\mathcal{B}[\![\mathtt{B}]\!] : \mathbb{M} \to \mathbb{B}_{\lightning}$ results in a runtime error if the arithmetic evaluation results in a runtime error.

$$\mathcal{B}[\![\mathtt{B}]\!] : \mathbb{M} \to \mathbb{B}_{\lightning}$$

$$\mathcal{B}[\![\mathtt{tt}]\!]m \triangleq \mathtt{tt} \tag{6.5}$$

$$\mathcal{B}[\![\mathtt{ff}]\!]m \triangleq \mathtt{ff} \tag{6.6}$$

$$\mathcal{B}[\![\neg \mathtt{B}_1]\!]m \triangleq \begin{cases} \lightning & \text{if } \mathcal{B}[\![\mathtt{B}_1]\!]m = \lightning \\ \neg \mathcal{B}[\![\mathtt{B}_1]\!]m & \text{otherwise} \end{cases} \tag{6.7}$$

$$\mathcal{B}[\![\mathtt{B}_1 \diamond \mathtt{B}_2]\!]m \triangleq \begin{cases} \lightning & \text{if } \mathcal{B}[\![\mathtt{B}_1]\!]m = \lightning \text{ or } \mathcal{B}[\![\mathtt{B}_2]\!]m = \lightning \\ \mathcal{B}[\![\mathtt{B}_1]\!]m \diamond \mathcal{B}[\![\mathtt{B}_2]\!]m & \text{otherwise} \end{cases} \tag{6.8}$$

$$\mathcal{B}[\![\mathtt{A}_1 \diamond \mathtt{A}_2]\!]m \triangleq \begin{cases} \lightning & \text{if } \mathcal{A}[\![\mathtt{A}_1]\!]m = \lightning \text{ or } \mathcal{A}[\![\mathtt{A}_2]\!]m = \lightning \\ \mathcal{A}[\![\mathtt{A}_1]\!]m \diamond \mathcal{A}[\![\mathtt{A}_2]\!]m & \text{otherwise} \end{cases} \tag{6.9}$$

6.2.2. Reachability semantics

The *state of a program* is represented by the values of the variables, and we define program states accordingly: $m \in \mathbb{S} \triangleq \mathbb{M}$. Observe that while in this introductory chapter program states and memories coincide, this will change in Chapter 7. We define program behaviours as sets of states, namely $M \in \mathbb{D} \triangleq \wp(\mathbb{S})$. The *statement reachability semantics* $\mathcal{S}[\![\mathtt{S}]\!] : \mathbb{D} \to \mathbb{D}$ is a function that, given a set of states before the statement (i.e., a *precondition*), associates the set of reachable states after the execution of the statement S (i.e., a *postcondition*). The semantics is *deterministic,* namely it associates each input state with at most one output state. Nondeterminism introduces complexity, and we will incorporate it in our language later, in Chapter 7. We now proceed to define by *structural induction* the semantics $\mathcal{S}[\![\mathtt{S}]\!]$. We start with the `skip` statement, which does not modify the input state.

$$\mathcal{S}[\![\mathtt{skip}]\!]M \triangleq M \tag{6.10}$$

The statements `x = A` has the effect to assign the variable `x` to the arithmetic evaluation of `A`. Since the evaluation can result in a runtime error, the states that present a division by zero are simply ignored in the postcondition.

$$\mathcal{S}[\![\mathtt{x = A}]\!]M \triangleq \{\, m[\mathtt{x} \leftarrow \mathcal{A}[\![\mathtt{A}]\!]m] \mid m \in M \wedge \mathcal{A}[\![\mathtt{A}]\!]m \neq \lightning \,\} \tag{6.11}$$

Example 6.2 (Assignment with division by zero)
Consider the following program that assigns `x` to 2 divided by `y`: `x = 2/y`. As input we consider two states: one where `y` is 2, and another where `y` is 0. While the first state leads to a reachable state where `x` is 1, the second one results in a runtime error that is filtered out in the output reachable states.

$$\mathcal{S}[\![\mathtt{x = 2/y}]\!]\{\{\mathtt{x} \mapsto 0, \mathtt{y} \mapsto 2\}, \{\mathtt{x} \mapsto 0, \mathtt{y} \mapsto 0\}\} = \{\{\mathtt{x} \mapsto 1, \mathtt{y} \mapsto 2\}\}$$

Statement composition composes the semantics of the two individual state-

ments.

$$\mathcal{S}[\![\mathtt{S}_1\,;\,\mathtt{S}_2]\!]M \triangleq \mathcal{S}[\![\mathtt{S}_2]\!](\mathcal{S}[\![\mathtt{S}_1]\!]M) \tag{6.12}$$

If statements filter the input states according to the statement condition, execute the two branches individually, and then join the reachable states at the end. Observe that the boolean evaluation of the condition can result in runtime errors, so that, similarly to assignments, states that result in a division by zero during the evaluation are filtered out. We rely on a helper function $\mathtt{test}[\![\mathtt{B}]\!] : \mathbb{D} \to \mathbb{D}$ that computes the reachable states after a condition.

$$\mathtt{test}[\![\mathtt{B}]\!]M \triangleq \{\, m \in M \mid \mathcal{B}[\![\mathtt{B}]\!]m = \mathtt{tt} \,\} \tag{6.13}$$

$$\mathcal{S}[\![\mathtt{if\ (B)\ S}_t\ \mathtt{else\ S}_e]\!]M \triangleq \mathcal{S}[\![\mathtt{S}_t]\!](\mathtt{test}[\![\mathtt{B}]\!]m) \cup \mathcal{S}[\![\mathtt{S}_e]\!](\mathtt{test}[\![\neg\mathtt{B}]\!]m) \tag{6.14}$$

While statements are more complex than all the others. Their semantics is expressed as a least fixpoint whose existence is guaranteed by Thm. 2.1 and Thm. 2.2 (see Section 2.3).

$$\begin{aligned}\mathcal{S}[\![\mathtt{while\ (B)\ S}_b]\!]M &\triangleq \mathtt{test}[\![\neg\mathtt{B}]\!](\mathrm{lfp}\,F) \\ \textbf{where } F(M_1) &\triangleq M \cup \mathcal{S}[\![\mathtt{S}_b]\!](\mathtt{test}[\![\mathtt{B}]\!]M_1)\end{aligned} \tag{6.15}$$

We observe that $\mathcal{S}[\![\mathtt{S}]\!]$ is monotonic and continuous, and it is furthermore an operator over the complete lattice $(\mathbb{D}, \subseteq, \cup, \cap, \emptyset, \mathbb{S})$. This implies that we can apply both Tarski's fixpoint theorem (Thm. 2.1) or Kleene's fixpoint theorem (Thm. 2.2) to observe that $\mathcal{S}[\![\mathtt{S}]\!]$ is well-defined. In particular, Kleene's theorem draws an insightful parallel with the iterative nature of while statements. In fact, we can observe that $F^0(\emptyset) = \emptyset$, and that $F^1(\emptyset) = M$, namely $F^1(\emptyset)$ is the set of reachable states *before* entering the loop. Then, $F^2(\emptyset) = M \cup \mathcal{S}[\![\mathtt{S}_t]\!]\mathtt{test}[\![\mathtt{B}]\!]M$, which is the set of reachable states at the loop head after at most one iteration of the loop. In general, $F^n(\emptyset)$ corresponds to the set of reachable states at the loop head after at most $n-1$ iterations. Then, the least fixpoint reachable at the loop head is exactly $\bigcup_{i\in\mathbb{N}} F^i(\emptyset)$, which corresponds to executing the loop an arbitrary number of times. To conclude, the states that exit the loop are those that do not satisfy the boolean condition $\mathtt{B}$, namely $\mathtt{test}[\![\neg\mathtt{B}]\!](\bigcup_{i\in\mathbb{N}} F^i(\emptyset))$.

Example 6.3 (Fixpoint semantics)
Consider the program in Example 6.1, which computes the factorial of n. Consider as initial state $\{\texttt{n} \mapsto 3, \texttt{factorial} \mapsto 0, \texttt{i} \mapsto 0\}$, so that $\{\{\texttt{n} \mapsto 3, \texttt{factorial} \mapsto 1, \texttt{i} \mapsto 2\}\}$ is the set of reachable states before entering the while loop. The Kleene's iterates are given by the following:

- $F^0(\emptyset) = \emptyset$.
- $F^1(\emptyset) = \{\{\texttt{n} \mapsto 3, \texttt{factorial} \mapsto 1, \texttt{i} \mapsto 2\}\}$.
- $F^2(\emptyset) = \{\{\texttt{n} \mapsto 3, \texttt{factorial} \mapsto 1, \texttt{i} \mapsto 2\}, \{\texttt{n} \mapsto 3, \texttt{factorial} \mapsto 2, \texttt{i} \mapsto 3\}\}$.
- $F^3(\emptyset) = \{\{\texttt{n} \mapsto 3, \texttt{factorial} \mapsto 1, \texttt{i} \mapsto 2\}, \{\texttt{n} \mapsto 3, \texttt{factorial} \mapsto 2, \texttt{i} \mapsto 3\}, \{\texttt{n} \mapsto 3, \texttt{factorial} \mapsto 6, \texttt{i} \mapsto 4\}\}$.
- $\forall i \geq 4 : F^i(\emptyset) = F^3(\emptyset)$, so that $\operatorname{lfp} F = \{\{\texttt{n} \mapsto 3, \texttt{factorial} \mapsto 1, \texttt{i} \mapsto 2\}, \{\texttt{n} \mapsto 3, \texttt{factorial} \mapsto 2, \texttt{i} \mapsto 3\}, \{\texttt{n} \mapsto 3, \texttt{factorial} \mapsto 6, \texttt{i} \mapsto 4\}\}$.

Then, the negation of the boolean condition `i <= n` is `i > n`, so that the resulting reachable states after the while loop is $\{\{\texttt{n} \mapsto 3, \texttt{factorial} \mapsto 6, \texttt{i} \mapsto 4\}\}$. We observe that, as we expected, the variable `factorial` stores the factorial of `n`.

While sometimes we can reach the least fixpoint of the function F after a finite number of iterations, in general this is not the case. In fact, as we show in the following example, we sometimes have to pass to the limit to obtain the least fixpoint.

Example 6.4 (Infinite Kleene's iterations)
Consider the following program:

```
1 n = 0;
2 while (tt) {
3   n = n + 1;
4 }
```

There are infinitely many different Kleene iterates of the function F. The least

fixpoint is given by the following:

$$\operatorname{lfp} F = \bigcup_{i \in \mathbb{N}} F^i(\emptyset) = \{\{\mathtt{n} \mapsto i\} \mid i \in \mathbb{N}\}$$

While the semantics of statements is a function from a set of states to another set of states, the semantics of a program $\mathtt{P} := \mathtt{S}$ is simply a set of states: $\mathbb{S}[\![\mathtt{P}]\!] \in \mathbb{D}$. We define the set of *initial states* $I \in \mathbb{D}$ as the set of all possible program memories:

$$I \triangleq \{m \in \mathbb{M}\} \tag{6.16}$$

Then, we obtain the *program reachability semantics* $\mathbb{S}[\![\mathtt{P}]\!] \in \mathbb{D}$ of $\mathtt{P} := \mathtt{S}$ by applying the statement semantics to the set of initial states.

$$\mathbb{S}[\![\mathtt{P}]\!] \triangleq \mathbb{S}[\![\mathtt{S}]\!]I \tag{6.17}$$

6.2.3. Trace semantics

While the reachability semantics presented in the previous section is well-suited to concisely represent the set of reachable states, it is not the most informative description of programs behaviour. In fact, the *trace semantics* that we describe in this section is strictly more expressive than the reachability semantics. We introduce the trace semantics because, as shown in Section 6.3, formal study of program properties necessitates reasoning based on the most informative semantics. In particular, the reachability semantics does not capture aspects of programs execution such as *nontermination*. When designing analyses, one has to choose the simplest semantics that can still infer all properties of interest. In the case of the analysis presented in Chapter 7, the reachability semantics is indeed sufficient. The trace semantics we present in this section is *stateful*, namely the state of the program is described in the elements of the trace, rather than the transitions. Alternative *stateless trace semantics* have been proposed [5, Chapter 6].

A *trace* is a sequence of pairs of program labels and states, namely a sequence $(\ell_1, m_1), (\ell_2, m_2), \ldots$. A trace is *finite* if it is an element of $\mathbb{T}^* \triangleq (\mathbb{L} \times \mathbb{S})^*$ or *infinite* if it is an element of $\mathbb{T}^\infty \triangleq (\mathbb{L} \times \mathbb{S})^\infty$. We denote the *empty trace* as $\varepsilon \in \mathbb{T}^*$. We define the set of *non-empty finite traces* as $\mathbb{T}^+ \triangleq \mathbb{T}^* \backslash \{\varepsilon\}$, and the set of *all traces* as $\mathbb{T}^{*\infty} \triangleq \mathbb{T}^* \cup \mathbb{T}^\infty$.

Let $t = (\ell_0, m_0), \ldots, (\ell_n, m_n) \in \mathbb{T}^*$ and $t' = (\ell'_0, m'_0), (\ell'_1, m'_1), \cdots \in \mathbb{T}^{*\infty}$ such that $\ell_n = \ell'_0$ and $m_n = m'_0$. Then, the *trace junction* operator is defined as follows:

$$t \frown t' \triangleq (\ell_0, m_0), \ldots, (\ell_n, m_n), (\ell'_1, m'_1), \ldots \tag{6.18}$$

Observe that $t \frown t'$ is undefined if $\ell_n \neq \ell'_0$ or $m_n \neq m'_0$. The trace junction operator is defined as $t \frown t' \triangleq t$ if $t \in \mathbb{T}^\infty$. Furthermore, we denote the length of a trace t as $|t|$. Observe that $|t| = \infty$ if $t \in \mathbb{T}^\infty$.

We now define by structural induction the *transition relation of statements*, namely a relation that describes the possible transitions from a pair (ℓ, m) to another induced by a statement $\mathtt{S}$. Note that it is a small-step operational semantics [120].

$$\tau[\![\mathtt{S}]\!] \in \wp((\mathbb{L} \times \mathbb{S}) \times (\mathbb{L} \times \mathbb{S}))$$

$$\tau[\![{}^{\ell_1}\mathtt{skip}^{\ell_2}]\!] \triangleq \{\, ((\ell_1, m), (\ell_2, m)) \mid m \in \mathbb{M} \,\} \tag{6.19}$$

$$\tau[\![{}^{\ell_1}\mathtt{x = A}^{\ell_2}]\!] \triangleq \{\, ((\ell_1, m), (\ell_2, m[\mathtt{x} \leftarrow \mathcal{A}[\![\mathtt{A}]\!]m])) \mid m \in \mathbb{M}, \mathcal{A}[\![\mathtt{A}]\!]m \neq \lightning \,\} \tag{6.20}$$

$$\tau[\![{}^{\ell_1}\mathtt{S}_1\,;\,{}^{\ell_2}\mathtt{S}_2^{\ell_3}]\!] \triangleq \tau[\![{}^{\ell_1}\mathtt{S}_1{}^{\ell_2}]\!] \cup \tau[\![{}^{\ell_2}\mathtt{S}_2{}^{\ell_3}]\!] \tag{6.21}$$

$$\begin{aligned}\tau[\![{}^{\ell_1}\mathtt{if\ (B)}\ {}^{\ell_2}\mathtt{S}_t{}^{\ell_3}\ \mathtt{else}\ {}^{\ell_4}\mathtt{S}_e{}^{\ell_5}\mathtt{fi}^{\ell_6}]\!] \triangleq\ & \{\, ((\ell_1, m), (\ell_2, m)) \mid m \in \mathbb{M}, \mathcal{B}[\![\mathtt{B}]\!]m = \mathtt{tt} \,\} \cup \\ & \{\, ((\ell_1, m), (\ell_4, m)) \mid m \in \mathbb{M}, \mathcal{B}[\![\mathtt{B}]\!]m = \mathtt{ff} \,\} \cup \\ & \tau[\![{}^{\ell_2}\mathtt{S}_t{}^{\ell_3}]\!] \cup \tau[\![{}^{\ell_4}\mathtt{S}_e{}^{\ell_5}]\!] \cup \\ & \{\, ((\ell_3, m), (\ell_6, m)) \mid m \in \mathbb{M} \,\} \cup \\ & \{\, ((\ell_5, m), (\ell_6, m)) \mid m \in \mathbb{M} \,\}\end{aligned} \tag{6.22}$$

$$\begin{aligned}\tau[\![{}^{\ell_1}\mathtt{while}\ {}^{\ell_2}\mathtt{(B)}\ {}^{\ell_3}\mathtt{S}_b{}^{\ell_4}\mathtt{od}^{\ell_5}]\!] \triangleq\ & \{\, ((\ell_1, m), (\ell_2, m)) \mid m \in \mathbb{M}, \,\} \cup \\ & \{\, ((\ell_2, m), (\ell_3, m)) \mid m \in \mathbb{M}, \mathcal{B}[\![\mathtt{B}]\!]m = \mathtt{tt} \,\} \cup \\ & \tau[\![{}^{\ell_3}\mathtt{S}_b{}^{\ell_4}]\!] \cup \\ & \{\, ((\ell_4, m), (\ell_2, m)) \mid m \in \mathbb{M}, \,\} \cup \\ & \{\, ((\ell_2, m), (\ell_5, m)) \mid m \in \mathbb{M}, \mathcal{B}[\![\mathtt{B}]\!]m = \mathtt{ff} \,\}\end{aligned} \tag{6.23}$$

Using the definition of $\tau[\![\mathtt{S}]\!]$, it is possible to define the *maximal trace semantics*, which collects all the complete execution traces of a program along with the infinite traces. Let $\mathtt{P} := \mathtt{S}$.

$$
\begin{aligned}
&\mathcal{S}_\infty[\![\mathtt{P}]\!] \in \wp(\mathbb{T}^{*\infty}) \\
&\mathcal{S}_\infty[\![\mathtt{P}]\!] \triangleq \{\, (\ell_1, m_1), \ldots, (\ell_n, m_n) \in \mathbb{T}^* \mid \\
&\qquad\qquad \forall i < n : ((\ell_i, m_i), (\ell_{i+1}, m_{i+1})) \in \tau[\![\mathtt{S}]\!] \\
&\qquad\qquad \text{and } \nexists (\ell, m) : ((\ell_n, m_n), (\ell, m)) \in \tau[\![\mathtt{S}]\!] \,\} \\
&\qquad \cup \{\, (\ell_1, m_1), \cdots \in \mathbb{T}^\infty \mid \forall i \geq 1 : ((\ell_i, m_i), (\ell_{i+1}, m_{i+1})) \in \tau[\![\mathtt{S}]\!] \,\}
\end{aligned}
\tag{6.24}
$$

Observe that it is possible to give a fixpoint characterization of the maximal trace semantics [121]. Furthermore, the reachability semantics described in Section 6.2.3 is an *abstraction* of the maximal trace semantics, namely the former can be inferred from the latter by ignoring some information. There are numerous semantics that can be organized into a hierarchy of successive abstractions [121].

6.3. Program properties

We are interested in classifying programs as "correct" or "incorrect" with respect to a certain specification, and in order to do this we mathematically formalize these intuitive concepts. *Properties* are specified by their *extension*, that is, the set of elements that have such property. Therefore, if X is an universe, properties of X are elements of $\wp(X)$. We first study *properties of traces* (trace properties), so that these types of properties are elements of $\wp(\mathbb{T}^{*\infty})$, i.e. sets of traces. Trace properties are sometimes simply referred as *properties*, and we describe them in detail in the following section. As we will discuss, they can be generalized and extended to *hyperproperties* [50], that are properties of program semantics.

6.3.1. Trace properties

In this section, we study two classes of trace properties, namely *safety* and *liveness*, and we will see that every trace property can be expressed as the conjunction of a safety and a liveness property.

Safety properties are the class of trace properties that ensure that certain undesirable conditions do not occur during the execution of a program, and are usually described as the class of properties stating that "something bad never

happens." They include extensively studied properties, such as *absence of runtime errors* and *partial correctness* [122] (i.e., all the terminating computations yield correct results). Informally, a trace property is a safety property such that, when it does not hold, it admits a *finite counter-example prefix trace*. We now give the formal definition of the class of safety properties.

Definition 6.1 (Safety property)
A trace property $P \in \wp(\mathbb{T}^{*\infty})$ is a *safety property* if and only if:

$$\forall T \in \wp(\mathbb{T}^{*\infty}), T \not\subseteq P \implies \\ \exists t \in \mathbb{T}^* : \exists t' \in \mathbb{T}^{*\infty} : t \frown t' \in T : \forall t'' \in \mathbb{T}^* : t \frown t'' \notin P$$

Example 6.5 (Safety property)
Let $S \in \wp(\mathbb{L} \times \mathbb{S})$ be a property of labels-states pairs. Then, $S^\infty \in \mathbb{T}^\infty$ is a safety property. If S is a set of "good" desirable states, then S^∞ is the safety property expressing that no "bad" state is ever reached.

Liveness properties are the class of trace properties that ensure that a certain desirable condition is eventually met, and they are usually described as properties stating that "something good eventually happens." Informally, a liveness property is a trace property such that any finite execution may be extended into a correct one. This implies that liveness properties do not admit finite counterexamples. The canonical example of liveness property is *termination*, which states that every computation eventually terminates. Observe that since liveness properties do not admit finite counterexamples, it is not possible to use testing to disprove a liveness property.

Definition 6.2 (Liveness property)
A trace property $P \in \wp(\mathbb{T}^{*\infty})$ is a *liveness property* if and only if:

$$\forall t \in \mathbb{T}^* : \exists t' \in \mathbb{T}^{*\infty} : t \frown t' \in P$$

Example 6.6 (Liveness property)
$\mathbb{T}^*$ is a liveness property. More precisely, it describes termination.

Observe that there are other definitions of safety and liveness properties in terms of topologically closed and dense sets [5, Chapter 14]. As proved by Alpern and Schneider [122], safety and liveness properties are sufficient to fully describe all trace properties. In fact, every trace property can be expressed as the intersection of a safety and a liveness property.

Theorem 6.1 (Trace properties as conjunction of safety and liveness [122])
Let $P \in \wp(\mathbb{T}^{*\infty})$ be a trace property. Then, there exist a safety property S and a liveness property L such that:

$$P = S \cap L$$

Example 6.7 (Total correctness)
Total correctness states that all the computations yield correct results. It is expressed as the conjunction of partial correctness (i.e., all the terminating computations yield correct results), and termination (i.e., all the computations are finite).

6.3.2. Hyperproperties

While trace properties have been extensively studied in the literature, they are not sufficiently expressive to describe some classes of program behaviours. The canonical example is *noninterference* [123, 124, 125], which requires public output data of a program not to depend on private input data. To express noninterference, it is necessary to compare different executions of the program, so that trace properties are not sufficient to express it. To overcome this limitation, the framework of *hyperproperties* [50] has been introduced. Hyperproperties, sometimes referred as *program properties* or *relational properties*, are sets of program semantics, and therefore elements of $\wp(\wp(\mathbb{T}^{*\infty}))$. They increase the expressiveness of trace properties by being able to relate different program executions. According to our definition of properties (a property is the set of elements that have the property), it would

be more appropriate to refer to hyperproperties as program properties. Nevertheless, the term hyperproperty has been widely used in the literature and it is well-understood, so that we use it as well in this work. Similarly to trace properties, hyperproperties can be divided in *hypersafety* and *hyperliveness*.

Hypersafety properties extend safety trace properties by allowing the counterexample observation to be a finite set of finite traces, rather than a single finite trace. Informally, hypersafety properties can be defined as those hyperproperties such that, if the property does not hold, then it admits a finite set of finite traces as counterexample. Given $T_1, T_2 \in \wp(\mathbb{T}^{*\infty})$, we say that T_2 *extends* T_1 and note $T_1 \leq T_2$ if and only if the following holds:

$$T_1 \leq T_2 \overset{\triangle}{\iff} \forall t \in T_1 : \exists t' \in \mathbb{T}^{*\infty} : t \frown t' \in T_2 \tag{6.25}$$

Definition 6.3 (Hypersafety property)
A hyperproperty $\mathscr{P} \in \wp(\wp(\mathbb{T}^{*\infty}))$ is a *hypersafety property* if and only if:

$$\forall T_1 \in \wp(\mathbb{T}^{*\infty}) : T_1 \notin \mathscr{P} \implies \exists M \in \wp(\mathbb{T}^*) : \begin{cases} M \text{ is finite} \wedge \\ M \leq T_1 \wedge \\ \forall T_2 \in \wp(\mathbb{T}^{*\infty}) : M \leq T_2 \implies T_2 \notin \mathscr{P} \end{cases}$$

While the definition requires the counterexample to be finite, it can be arbitrarily large. For this reason, we introduce *k-hypersafety properties*, which bound the maximum size of the counterexample to k traces for $k \geq 1$.

Definition 6.4 (k-hypersafety property)
A hyperproperty $\mathscr{P} \in \wp(\wp(\mathbb{T}^{*\infty}))$ is a *k-hypersafety property* if and only if:

$$\forall T_1 \in \wp(\mathbb{T}^{*\infty}) : T_1 \notin \mathscr{P} \implies \exists M \in \wp(\mathbb{T}^*) : \begin{cases} |M| \leq k \wedge \\ M \leq T_1 \wedge \\ \forall T_2 \in \wp(\mathbb{T}^{*\infty}) : M \leq T_2 \implies T_2 \notin \mathscr{P} \end{cases}$$

Remark 6.2 (Safety properties as 1-hypersafety)
All safety properties are 1-hypersafety, because counter-examples consist of only one finite trace. This implies that hypersafety properties generalize safety properties.

Noninterference informally requires public output data not to depend on private input data, and we now formalize this property as a 2-hypersafety property. First, we need to partition the variables between *public variables* $\mathbb{V}_{\mathtt{pub}}$ and *private variables* $\mathbb{V}_{\mathtt{priv}}$, and $\mathbb{V} = \mathbb{V}_{\mathtt{pub}} \cup \mathbb{V}_{\mathtt{priv}}$. Public variables are those that can be observed by the users of the system, while private variables are those that hold the values of secret data which should not be leaked. For $m_1, m_2 \in \mathbb{M}$, we write $m_1 =_{\mathtt{pub}} m_2$ if and only if the two memories agree on the values of the public variables:

$$m_1 =_{\mathtt{pub}} m_2 \stackrel{\triangle}{\iff} \forall \mathtt{x} \in \mathbb{V}_{\mathtt{pub}} : m_1(\mathtt{x}) = m_2(\mathtt{x}) \tag{6.26}$$

Then, we can finally define noninterference as the set of sets of traces such that traces that agree on the values of public variables at the beginning, agree on the values of public variables at the end.

Definition 6.5 (Noninterference)

$$\mathcal{NI} \in \wp(\wp(\mathbb{T}^*))$$
$$\mathcal{NI} \triangleq \{\, T \in \wp(\mathbb{T}^*) \mid \forall t = (\ell_1, m_1), \ldots, (\ell_n, m_n),\, t' = (\ell'_1, m'_1), \ldots, (\ell'_{n'}, m'_{n'}) \in T : m_1 =_{\mathtt{pub}} m'_1 \implies m_n =_{\mathtt{pub}} m'_{n'} \,\}$$

Noninterference is a 2-hypersafety property, as it is sufficient to observe two traces that agree on public variables at the beginning, but differ in the value of at least one public variable at the end to disprove $\mathcal{NI}$. For the rest of this section, if not otherwise specified, we let $\mathbb{V}_{\mathtt{pub}} = \{\mathtt{x}_{\mathtt{pub}}\}$ and $\mathbb{V}_{\mathtt{priv}} = \{\mathtt{x}_{\mathtt{priv}}\}$.

Example 6.8 (Explicit flow)
Consider the following program:

$$\mathtt{x}_{\mathtt{pub}} = \mathtt{x}_{\mathtt{priv}}$$

The program is *interferent*, because by modifying the private input of the program, the public output changes. A dependency that is propagated through assignments is known as an *explicit flow* [126]. Consider now the following program:

$$\mathtt{x_{pub}} = \mathtt{x_{priv}} - \mathtt{x_{priv}}$$

Even if it seems like the program is interferent, due to the public output variable being assigned to an expression involving private input data, this is not actually the case. In fact, the final value for $\mathtt{x_{pub}}$ is always 0, independently from the value of $\mathtt{x_{priv}}$. This implies that $\mathtt{x_{pub}}$ does not depend on the private input, which means that the program is noninterferent. Hence, the notion of noninterference is not purely syntactic but rather semantic, as it must take into account the values of variables and expressions.

Example 6.9 (Implicit flow)
Consider the following program:

$$\mathtt{if\ (x_{priv}\ ==\ 0)\ \{\ x_{pub} = 1\ \}\ else\ \{\ x_{pub} = 2\ \}}$$

At the end of the program, value of $\mathtt{x_{pub}}$ is 1 or 2 depending on the value of $\mathtt{x_{priv}}$. This implies that the program is interferent, even though the variable $\mathtt{x_{pub}}$ is not explicitly assigned to $\mathtt{x_{priv}}$. Dependencies that arise from the program control are known as *implicit flows* [126]. Often implicit flows are ignored [127], as they are considered less dangerous than explicit flows. Nevertheless, since we rely on a precise semantic definition of noninterference, in this work we do take them into account.

As we will see in Chapter 7, an important class of hyperproperties is the *subset-closed* hyperproperties. Informally, a hyperproperty is subset-closed when each subset of the set of traces that have the property, also has the property.

Definition 6.6 (Subset-closed hyperproperty)
Let $\mathscr{P} \in \wp(\wp(\mathbb{T}^{*\infty}))$ be a hyperproperty. Then, $\mathscr{P}$ is *subset-closed* if and only if

the following holds.

$$\forall T \in \wp(\mathbb{T}^{*\infty}) : T \in \mathscr{P} \implies (\forall T' \in \wp(\mathbb{T}^{*\infty}) : T' \subseteq T \implies T' \in \mathscr{P})$$

As it turns out, every hypersafety property is subset-closed.

Theorem 6.2 (Hypersafety properties are subset-closed [50])
Let $\mathscr{P} \in \wp(\wp(\mathbb{T}^{*\infty}))$ be a hypersafety property. Then, $\mathscr{P}$ is subset-closed.

A hyperproperty is a hyperliveness property if and only if, for each finite set of finite traces, it is possible to make the set respect the property by extending it with another set of traces.

Definition 6.7 (Hyperliveness property)
A hyperproperty $\mathscr{P} \in \wp(\wp(\mathbb{T}^{*\infty}))$ is a *hyperliveness property* if and only if:

$$\forall T_1 \in \wp(\mathbb{T}^*) : T_1 \text{ is finite} \implies \exists T_2 \in \wp(\mathbb{T}^{*\infty}) : T_1 \leq T_2 \ \wedge \ T_2 \in \mathscr{P}$$

Example 6.10 (Hyperliveness property)
We define *interference* as follows:

$$\mathscr{I} \in \wp(\wp(\mathbb{T}^*))$$
$$\mathscr{I} \triangleq \{\, T \in \wp(\mathbb{T}^*) \mid \exists t = (\ell_1, m_1), \ldots, (\ell_n, m_n), t' = (\ell'_1, m'_1), \ldots, (\ell'_{n'}, m'_{n'}) \in T : m_1 =_{\text{pub}} m'_1 \wedge m_n \neq_{\text{pub}} m'_{n'} \,\}$$

As it turns out, interference is a hyperliveness property. In fact, it is always possible to extend a finite set of finite traces with two interferent traces to make it respect $\mathscr{I}$.

In this work, we mainly focus on hypersafety properties. Nevertheless, we use hyperliveness properties to fully characterize the set of hyperproperties. In fact, similarly to the case of trace properties (see Thm. 6.1), any hyperproperty can be represented as the intersection of a hypersafety and a hyperliveness property.

Theorem 6.3 (Hyperproperties as conjunctions of hypersafety and hyperliveness [50])
Let $\mathscr{P} \in \wp(\wp(\mathbb{T}^{*\infty}))$ be a hyperproperty. Then, there exist a hypersafety property $\mathscr{S}$ and a hyperliveness property $\mathscr{L}$ such that:

$$\mathscr{P} = \mathscr{S} \cap \mathscr{L}$$

6.3.3. Undecidability of semantic program properties

While software engineering techniques extensively studied *syntactic program properties* (e.g. number of lines in each function, number of nested loops, etc.), *semantic program properties,* that is properties of the semantics of programs, are more challenging to be verified. Indeed, it is *impossible* to write a program that accepts as input another program and verifies its correctness, and this result is known as *Rice's undecidability theorem*.

Theorem 6.4 (Rice's Undecidability Theorem [22])
All non-trivial semantic properties of programs are undecidable.

Rice's theorem is a well-known impossibility result that states that if a semantic property is non-trivial (i.e., that is neither true nor false for every program), then there is no algorithm that can prove it. The result can be seen as an extension of Alan Turing's result on the undecidability of the halting problem [128]. Even if the result seems limiting, in the following section we show how it is nevertheless possible, under suitable hypotheses, to still prove some program properties.

6.4. Static analysis and abstract interpretation

Computer scientists in the field of *formal methods* have developed a rich set of techniques to elude Rice's theorem. This can be done by sacrificing either *completeness* (all true facts are provable), *soundness* (the conclusions about programs are always correct under suitable explicitly stated hypotheses), or *automation* (proofs are carried out by a computer). The main approaches to formal methods are the following:

Deductive methods. Deductive methods produce proofs of correctness, but ultimately require user interaction. This approach makes it possible to prove strong properties of programs, such as functional correctness with respect to a specification. Even if proof assistants help the user with hints and strategies to carry out a proof, this process cannot be ultimately fully automatized. Proof assistants such as COQ [129], LEAN [130], ISABELLE [131], and AGDA [132] are widely used in the field of deductive verification, and proof-oriented programming languages such as F* [133], DAFNY [134], WHYML [135], and SPARK [136] are designed to natively support formal software verification. The FRAMA-C [137] framework builds upon WHY3 [135] to allow deductive verification of user-annotated C code.

Symbolic execution. Symbolic execution techniques perform an abstract execution of programs by assuming symbolic variables for the unknown values, and propagate them during the analysis. The collected constraints are precise, and can be solved to determine if an arbitrary assertion is violated (e.g., absence of runtime errors). Since the number of feasible execution paths grows exponentially with the size of programs, symbolic execution techniques have sometimes to trade soundness for performance [23]. Examples of symbolic execution engines are ANGR [138], CRUCIBLE [139], BINSEC [140], and KLEE [141].

Model checking. Model checking restricts the verification problem to decidable fragments of languages [24] and produces correctness proofs automatically. Clarke et al. [25] apply *bounded model checking* to prove the correctness of ANSI-C programs. Their approach unwinds loops and function calls up to a threshold, which implies that behaviours beyond such a threshold are not considered. *Symbolic model checking* generalizes the approach to infinite but regular models [142]. CPACHECKER [143], ULTIMATEAUTOMIZER [144], and CBMC [145] are examples of model checkers for C.

Static analysis. Static analysis approaches analyze programs without the intervention of the user, and can automatically prove properties such as absence of runtime errors. Since the languages are not restricted to decidable fragments, the approach is necessarily *not complete*: correct programs can be classified

as dangerous, meaning that the analyzer can raise *false positives*. On the other hand, if the analyzer determines that a program is correct, then there is a strong mathematical guarantee about the fact that errors will not occur at runtime, namely there are *no false negatives*.

In this work, we rely on static analysis by *abstract interpretation* [26], which is a general theory of the approximation of formal program semantics. *Abstract interpreters*, i.e. analyzers that rely on abstract interpretation theory, run an abstract execution on the program and collect an overapproximation of the reachable states. In a single run, they consider all concrete executions, to which they necessarily add some spurious, hopefully irrelevant, ones. Since an abstract interpreter considers all reachable program states, if it finds a program to be error-free, then this proves that the program is correct. On the other hand, if the spurious executions turn out to be incorrect, an abstract interpreter can fail to prove that a correct program is indeed error-free. In this section, we propose a lightweight introduction to the theory of abstract interpretation. Various books and tutorials are available for an in-depth description of the topic [5, 146, 147].

The method we describe in what follows is suitable to prove safety properties, but can be adapted to prove liveness properties [148, 149, 150]. On the other hand, the verification of hyperproperties poses some challenges, and we discuss it further in Chapter 7.

6.4.1. Concrete and abstract elements

We now introduce the concepts of *concrete* and *abstract* elements, which are independent from program analysis. While in this section we reason on a generic set of concrete elements, in Section 6.4.3 we instantiate this concept to sets of memories, which are the concrete elements that we ultimately want to reason about.

The *concrete* set is a poset $(C, \leq)$, and the *abstract* set is a poset $(A, \sqsubseteq)$. The connection between the two sets is given by a *concretization function* $\gamma : A \to C$ that assigns meaning in terms of a concrete element to each abstract one.

Example 6.11 (Interval abstraction)
To study the concepts of abstract and concrete elements, we instantiate the concrete set to $(\wp(\mathbb{Z}), \subseteq)$. To abstract sets of integers, we use *intervals*, which

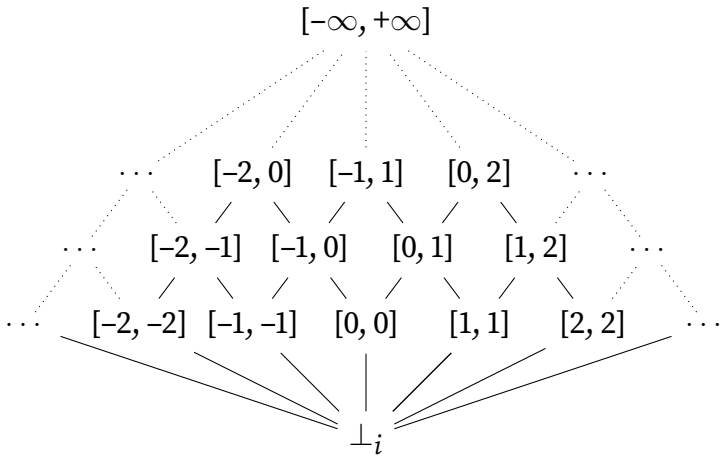


FIGURE 6.2. The interval complete lattice

are elements of the following set:

$$\mathbb{I} \triangleq \{[l, u] \mid l \in \mathbb{Z} \cup \{-\infty\}, u \in \mathbb{Z} \cup \{\infty\}, l \leq u\} \cup \bot_i \tag{6.27}$$

The partial order $\sqsubseteq_i$ for intervals is defined as follows:

$$\forall i : \bot_i \sqsubseteq_i i \tag{6.28}$$

$$\forall [l_1, u_1], [l_2, u_2] : [l_1, u_1] \sqsubseteq_i [l_2, u_2] \iff^{\triangle} l_2 \leq l_1 \wedge u_1 \leq u_2 \tag{6.29}$$

Then, the concretization function $\gamma_i : \mathbb{I} \to \wp(\mathbb{Z})$ associates an interval $[l, u]$ to the set of numbers that range from l to u:

$$\gamma_i(\bot_i) \triangleq \emptyset \tag{6.30}$$

$$\gamma_i([l, u]) \triangleq \{n \in \mathbb{Z} \mid l \leq n \leq u\} \tag{6.31}$$

A fundamental notion in abstract interpretation theory is *soundness*. An abstract element a is a *sound abstraction* of a concrete element c when the concretization of a is greater, with respect to the concrete partial order, than c. Intuitively, this captures the idea that the abstract element carries at least all the information in the concrete one (and possibly more), and that the abstract element can be used to soundly reason about the concrete one.

Definition 6.8 (Sound abstraction)
Let $c \in C$ and $a \in A$. Then, a is a *sound abstraction* of c if and only if:

$$c \leq \gamma(a)$$

Example 6.12 (Sound abstraction)
The interval $[0, 3]$ is a sound abstraction of $\{ 0, 2 \}$:

$$\{ 0, 2 \} \subseteq \{ 0, 1, 2, 3 \} = \gamma_i([0, 3])$$

If an abstract element a is not a sound abstraction of a concrete element c, then we say that a is an *unsound abstraction* of c.

Example 6.13 (Unsound abstraction)
The interval $[0, 1]$ is an unsound abstraction of $\{ 0, 2 \}$:

$$\{ 0, 2 \} \not\subseteq \{ 0, 1 \} = \gamma_i([0, 1])$$

Exactness is a strictly stronger notion than soundness. An abstract element a is an *exact abstraction* of a concrete element c in case its concretization is exactly equal to c.

Definition 6.9 (Exact abstraction)
Let $c \in C$ and $a \in A$. Then, a is an *exact abstraction* of c if and only if:

$$c = \gamma(a)$$

Intuitively, exact abstractions do not lose information about the concrete element that they represent. It is often impossible in static analysis to have an abstraction that exactly represents the concrete behaviour of a program, and for this reason we usually work with sound but inexact abstractions. The notions of soundness and exactness naturally extend to operators.

Definition 6.10 (Sound operator abstraction)
Let $f : C \to C$ be a concrete operator over C. Then, $f^\sharp : A \to A$ is a *sound* abstraction of f iff:

$$\forall a \in A : f(\gamma(a)) \leq \gamma(f^\sharp(a))$$

Definition 6.11 (Exact operator abstraction)
Let $f : C \to C$ be a concrete operator over C. Then, $f^\sharp : A \to A$ is an *exact* abstraction of f iff:

$$\forall a \in A : f(\gamma(a)) = \gamma(f^\sharp(a))$$

Sound operator abstraction is an important concept in abstract interpretation theory, as our ultimate goal is to replace operators that manipulate concrete elements (such as the reachability semantics $\mathcal{S}[\![\mathtt{S}]\!]$) with sound computable operators that reason on abstract elements.

Convention 6.1 (Abstract elements and operators)
Abstract elements and operators are often suffixed with the symbol $\sharp$ to distinguish them from the concrete ones.

6.4.2. The best abstraction: Galois connections

While there might be multiple sound abstractions of a concrete element c, it is interesting to study which one of those is the *best* one, namely the most precise sound abstraction of c. *Galois connections* formalize the correspondence between concrete elements and the abstract ones in case there is a best abstraction.

Definition 6.12 (Galois connection)
Let $(C, \leq)$ (the concrete set) and $(A, \sqsubseteq)$ (the abstract set) be two posets. Then, the pair (α, γ) of functions $\alpha : C \to A$ (the *upper adjoint*) and $\gamma : A \to C$ (the *lower ajoint*) is a *Galois connection* if and only if:

$$\forall c \in C : \forall a \in A : \alpha(c) \sqsubseteq a \iff c \leq \gamma(a)$$

In this case, we write $(C, \leq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (A, \sqsubseteq)$.

Example 6.14 (Galois connection)
Consider $(\wp(\mathbb{Z}), \subseteq)$ and $(\mathbb{I}, \sqsubseteq_i)$ as the concrete and the abstract sets. We define the abstraction function $\alpha_i : \wp(\mathbb{Z}) \to \mathbb{I}$ as follows:

$$\alpha_i(\emptyset) \triangleq \bot_i \tag{6.32}$$
$$\alpha_i(S) \triangleq [\min S, \max S] \tag{6.33}$$

Then, (α_i, γ_i) form a Galois connection $(\wp(\mathbb{Z}), \subseteq) \underset{\alpha_i}{\overset{\gamma_i}{\leftrightarrows}} (\mathbb{I}, \sqsubseteq_i)$. Let $S \in \wp(\mathbb{Z})$. The case $S \subseteq \gamma_i(\bot_i) \iff \alpha_i(S) \sqsubseteq_i \bot_i$ trivially holds, as $\gamma_i(\bot_i) = \emptyset$. Let $[l, u] \in \mathbb{I}$:

$$\begin{aligned} S \subseteq \gamma_i([l,u]) &\iff \forall n \in S : n \in \gamma_i([l,u]) \\ &\iff \forall n \in S : l \leq n \leq u \\ &\iff l \leq \min S \wedge \max S \leq u \\ &\iff [\min S, \max S] \sqsubseteq_i [l,u] \\ &\iff \alpha_i(S) \sqsubseteq_i [l,u] \end{aligned}$$

Galois connections are interesting because the lower adjoint (or *abstraction function*) gives the most precise sound abstraction for each concrete element.

Remark 6.3 (Best abstraction)
Let $(C, \leq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (A, \sqsubseteq)$. Then, $\forall c \in C$ it holds that $\alpha(c)$ is the *best abstraction*, namely the smallest (with respect to $\sqsubseteq$) sound abstraction of c.

The concept of best abstraction also extends to operators.

Remark 6.4 (Best operator abstraction)
Let $(C, \leq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (A, \sqsubseteq)$, and $f : C \to C$ be a concrete operator over C. Then, the best abstraction (with respect to $\sqsubseteq$) of f is given by $\alpha \circ f \circ \gamma$.

Example 6.15 (Best operator abstraction)
We want to overapproximate the addition between sets of integers defined as

follows:

$$S_1 + S_2 \triangleq \{\, n_1 + n_2 \mid n_1 \in S_1 \wedge n_2 \in S_2 \,\}$$

Since there is a Galois connection $(\wp(\mathbb{Z}), \subseteq) \underset{\alpha_i}{\overset{\gamma_i}{\leftrightarrows}} (\mathbb{I}, \sqsubseteq_i)$, we can apply Remark 6.4, and obtain that the best abstraction for the addition between sets of integers is given by the following:

$$\begin{aligned}
\bot_i +_i i &\triangleq \bot_i \\
i +_i \bot_i &\triangleq \bot_i \\
[l_1, u_2] +_i [l_2, u_2] &\triangleq \alpha_i(\gamma_i([l_1, u_2]) + \gamma_i([l_2, u_2])) \\
&= \alpha_i(\{\, n_1 \mid l_1 \leq n_1 \leq u_1 \,\} + \{\, n_2 \mid l_2 \leq n_2 \leq u_2 \,\}) \\
&= \alpha_i(\{\, n_1 + n_2 \mid l_1 \leq n_1 \leq u_1 \wedge l_2 \leq n_2 \leq u_2 \,\}) \\
&= [l_1 + l_2, u_1 + u_2]
\end{aligned}$$

Similarly to addition, we can apply Remark 6.4 and derive the best operator abstractions for subtraction $-_i$, multiplication $\times_i$, and division $/_i$. We also show how to derive the best operator abstraction for the set union and intersection.

$$\begin{aligned}
\bot_i \cup_i i &\triangleq i \\
i \cup_i \bot_i &\triangleq i \\
[l_1, u_1] \cup_i [l_2, u_2] &\triangleq \alpha_i(\gamma_i([l_1, u_2]) \cup \gamma_i([l_2, u_2])) \\
&= \alpha_i(\{\, n \mid l_1 \leq n \leq u_1 \vee l_2 \leq n \leq u_2 \,\}) \\
&= [\min(l_1, l_2), \max(u_1, u_2)] \\
\bot_i \cap_i i &\triangleq \bot_i \\
i \cap_i \bot_i &\triangleq \bot_i \\
[l_1, u_1] \cap_i [l_2, u_2] &\triangleq \alpha_i(\gamma_i([l_1, u_2]) \cap \gamma_i([l_2, u_2])) \\
&= \alpha_i(\{\, n \mid l_1 \leq n \leq u_1 \wedge l_2 \leq n \leq u_2 \,\}) \\
&= \begin{cases} [\max(l_1, l_2), \min(u_1, u_2)] & \text{if } \max(l_1, l_2) \leq \min(u_1, u_2) \\ \bot_i & \text{otherwise} \end{cases}
\end{aligned}$$

As it turns out, not every pair of concrete-abstract sets enjoys a Galois connection, so that there might not always exist an abstraction function α.

Example 6.16 (Absence of Galois connection)
Consider as concrete elements intervals over real numbers, and as abstract elements intervals over rational numbers. Some concrete elements, such as $[0, \sqrt{2}]$, do not have a smallest enclosing rational interval, hence, no α function can exist. This implies that there is no possible Galois connection between the two sets.

Observe that even if two sets do not enjoy a Galois connection, it is still possible to reason in terms of sound abstractions of the concrete elements. Indeed, as discussed in Section 6.4.3, numerous useful abstractions for program states do not enjoy a Galois connection with the concrete domain $\mathbb{D}$.

6.4.3. Static analysis and abstract domains

We now instantiate the concepts presented in the previous sections to program analysis. We first show how to define a *sound abstract semantics* of the WHILE language by using as running example the *interval abstract domain* [26]. The abstract semantics is a *computable* semantics that overapproximates the concrete semantics $\mathcal{S}[\![\mathtt{S}]\!]$. The abstract semantics yields an algorithm to effectively calculate an overapproximation of the reachable program states, and for this reason the terms *abstract semantics* and *analysis* are often used interchangeably. After introducing the interval abstract semantics, we formalize the concepts of *abstract value domain* and *abstract domain*, and we present different existing domains.

As running example, we first show how to overapproximate program states (i.e., sets of memories) by abstracting the values of each individual variable with an interval. To abstract sets of memories we use the *interval abstract domain*, which is defined as follows:

$$\mathbb{V}_{\mathbb{I}}^{\sharp} \triangleq (\mathbb{V} \to (\mathbb{I} \setminus \{\bot_i\})) \cup \{\bot_e\} \tag{6.34}$$

Each variable is assigned to an interval which overapproximates the set of possible values that can occur in the concrete semantics. If a variable is assigned to the empty interval $\bot_i$, all the other variables should be $\bot_i$ as well. To avoid multiple equivalent representations of the bottom state, we use a single representation for

bottom, namely $\bot_e$. The domain is ordered by the following partial order:

$$M_1^\sharp \sqsubseteq^\sharp_{\mathbb{V}_\square^\sharp} M_2^\sharp \overset{\triangle}{\iff} (M_1^\sharp = \bot_e) \vee (M_1^\sharp, M_2^\sharp \neq \bot_e \wedge \forall \mathrm{x} \in \mathbb{V} : M_1^\sharp(\mathrm{x}) \sqsubseteq_i M_2^\sharp(\mathrm{x})) \tag{6.35}$$

The concretization function is given by the following:

$$\begin{aligned} \gamma_{\mathbb{V}_\square^\sharp} &: \mathbb{V}_\square^\sharp \to \mathbb{D} \\ \gamma_{\mathbb{V}_\square^\sharp}(M^\sharp) &\triangleq \begin{cases} \emptyset & \text{if } M^\sharp = \bot_e \\ \{\, m \in \mathbb{M} \mid \forall \mathrm{x} \in \mathbb{V} : m(\mathrm{x}) \in \gamma_i(M^\sharp(\mathrm{x})) \,\} & \text{otherwise} \end{cases} \end{aligned} \tag{6.36}$$

Example 6.17 (Concretization of interval abstraction)
Consider the abstract element $\{\, \mathrm{x} \mapsto [-1, 1], \mathrm{y} \mapsto [0, 1] \,\}$. Its concretization corresponds to the following set of states:

$$\begin{aligned} \{ &\{\, \mathrm{x} \mapsto -1, \mathrm{y} \mapsto 0 \,\}, \{\, \mathrm{x} \mapsto 0, \mathrm{y} \mapsto 0 \,\}, \{\, \mathrm{x} \mapsto 1, \mathrm{y} \mapsto 0 \,\}, \\ &\{\, \mathrm{x} \mapsto -1, \mathrm{y} \mapsto 1 \,\}, \{\, \mathrm{x} \mapsto 0, \mathrm{y} \mapsto 1 \,\}, \{\, \mathrm{x} \mapsto 1, \mathrm{y} \mapsto 1 \,\} \} \end{aligned}$$

Remark 6.5 (Effective representation)
Observe that while $\mathbb{D}$ is not machine-representable (its elements can be infinite sets of memories), $\mathbb{V}_\square^\sharp$ admits an effective representation. This is because the set of variables is finite, and to each variable we associate an interval that consists of two finitely representable bounds.

We define an abstraction function for the interval abstract domain which leverages the abstraction function for intervals:

$$\begin{aligned} \alpha_{\mathbb{V}_\square^\sharp} &: \mathbb{D} \to \mathbb{V}_\square^\sharp \\ \alpha_{\mathbb{V}_\square^\sharp}(M) &\triangleq \begin{cases} \bot_e & \text{if } M = \emptyset \\ \lambda \mathrm{x} \in \mathbb{V}. \alpha_i(\{\, m(\mathrm{x}) \mid m \in M \,\}) & \text{otherwise} \end{cases} \end{aligned} \tag{6.37}$$

As it turns out, there is a Galois connection $(\mathbb{D}, \subseteq) \underset{\alpha_{\mathbb{V}_\square^\sharp}}{\overset{\gamma_{\mathbb{V}_\square^\sharp}}{\leftrightarrows}} (\mathbb{V}_\square^\sharp, \sqsubseteq^\sharp_{\mathbb{V}_\square^\sharp})$. By applying

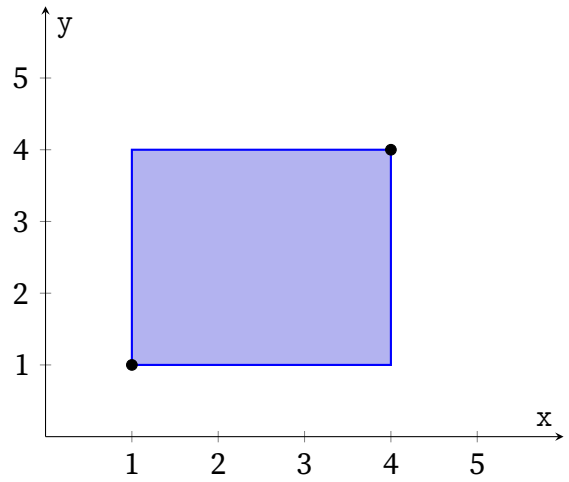


FIGURE 6.3. Interval abstraction of the concrete state $\{\{x \mapsto 1, y \mapsto 1\}, \{x \mapsto 4, y \mapsto 4\}\}$

the definition of $\alpha_{\mathbb{V}_{[]}^{\sharp}}$, we observe that the abstraction of the set of initial states I corresponds to the following:

$$\alpha_{\mathbb{V}_{[]}^{\sharp}}(I) = \lambda x \in \mathbb{V}.[-\infty, +\infty] \tag{6.38}$$

Remark 6.6 (Intervals as boxes)
Figure 6.3 shows a graphical representation of the interval abstraction for the concrete state $\{\{x \mapsto 1, y \mapsto 1\}, \{x \mapsto 4, y \mapsto 4\}\}$. The interval abstract domain overapproximates a set of points with the smallest enclosing rectangle containing them.

Observe that since each variable is abstracted individually, the interval domain is not expressive enough to capture the relations between variables.

Example 6.18 (Loss of information in the interval domain)
Consider the concrete state $\{\{x \mapsto 1, y \mapsto 1\}, \{x \mapsto 4, y \mapsto 4\}\}$. The best interval abstraction (given by $\alpha_{\mathbb{V}_{[]}^{\sharp}}$) is $\{x \mapsto [1, 4], y \mapsto [1, 4]\}$, which loses the relational information $x = y$.

We also define join and meet operators to merge and intersect two abstract

maps which leverage the join and meet operators over intervals (see Example 6.15).

$$\cup^\sharp_{\mathbb{V}^\sharp_{[]}} : \mathbb{V}^\sharp_{[]} \times \mathbb{V}^\sharp_{[]} \to \mathbb{V}^\sharp_{[]}$$

$$M_1^\sharp \cup^\sharp_{\mathbb{V}^\sharp_{[]}} M_2^\sharp \triangleq \begin{cases} M_2^\sharp & \text{if } M_1^\sharp = \bot_e \\ M_1^\sharp & \text{if } M_2^\sharp = \bot_e \\ \lambda \mathrm{x} \in \mathbb{V}. M_1^\sharp(\mathrm{x}) \cup_i M_2^\sharp(\mathrm{x}) & \text{otherwise} \end{cases} \tag{6.39}$$

$$\cap^\sharp_{\mathbb{V}^\sharp_{[]}} : \mathbb{V}^\sharp_{[]} \times \mathbb{V}^\sharp_{[]} \to \mathbb{V}^\sharp_{[]}$$

$$M_1^\sharp \cap^\sharp_{\mathbb{V}^\sharp_{[]}} M_2^\sharp \triangleq \begin{cases} \bot_e & \text{if } M_1^\sharp = \bot_e \text{ or } M_2^\sharp = \bot_e \\ \bot_e & \text{if } \exists \mathrm{x} : M_1^\sharp(\mathrm{x}) \cap_i M_2^\sharp(\mathrm{x}) = \bot_i \\ \lambda \mathrm{x} \in \mathbb{V}. M_1^\sharp(\mathrm{x}) \cap_i M_2^\sharp(\mathrm{x}) & \text{otherwise} \end{cases} \tag{6.40}$$

Abstract arithmetic expression evaluation

Now that we have described the abstract domain we use for overapproximating sets of memories, we define a sound abstraction for the arithmetic evaluation of expressions. The abstract arithmetic evaluation of expressions, denoted as $\mathcal{A}^\sharp_{\mathbb{V}^\sharp_{[]}}[\![\mathtt{A}]\!] : \mathbb{V}^\sharp_{[]} \to \mathbb{I}$, is sound with respect to the following criterion.

Theorem 6.5 (Soundness of abstract interval arithmetic expression evaluation)

$$\forall M^\sharp \in \mathbb{V}^\sharp_{[]} : \forall m \in \gamma_{\mathbb{V}^\sharp_{[]}}(M^\sharp) : \mathcal{A}[\![\mathtt{A}]\!] m \in \gamma_i(\mathcal{A}^\sharp_{\mathbb{V}^\sharp_{[]}}[\![\mathtt{A}]\!] M^\sharp)$$

The theorem states that, for each concrete memory overapproximated by an abstract memory, the result of the evaluation in the concrete is overapproximated by the result of the abstract evaluation. We can finally define the abstract evaluation of expressions for the interval domain by relying on the operators $+_i, -_i, \times_i, /_i$ (see Example 6.15).

$$\mathcal{A}^\sharp_{\mathbb{V}^\sharp_{[]}}[\![\mathtt{n}]\!]M^\sharp \triangleq [\mathtt{n}, \mathtt{n}] \tag{6.41}$$

$$\mathcal{A}^\sharp_{\mathbb{V}^\sharp_{[]}}[\![\mathtt{x}]\!]M^\sharp \triangleq M^\sharp(\mathtt{x}) \tag{6.42}$$

$$\mathcal{A}^\sharp_{\mathbb{V}^\sharp_{[]}}[\![\mathtt{A}_1 \diamond \mathtt{A}_2]\!]M^\sharp \triangleq \mathcal{A}^\sharp_{\mathbb{V}^\sharp_{[]}}[\![\mathtt{A}_1]\!]M^\sharp \diamond_i \mathcal{A}^\sharp_{\mathbb{V}^\sharp_{[]}}[\![\mathtt{A}_2]\!]M^\sharp \tag{6.43}$$

Example 6.19 (Abstract arithmetic evaluation)

$$\mathcal{A}^\sharp_{\mathbb{V}^\sharp_{[]}}[\![\mathtt{x} + 1]\!]\{\,\mathtt{x} \mapsto [0, 10]\,\} = [0, 10] +_i [1, 1] = [1, 11]$$

Abstract test evaluation

The concrete semantics relies on tests to filter the states that respect a boolean condition. To overapproximate tests, a practical solution is to define ad-hoc functions for simple and common cases, and revert to a fallback sound implementation for more complex conditions. The abstract test operator $\mathtt{test}^\sharp_{\mathbb{V}^\sharp_{[]}}[\![\mathtt{B}]\!] : \mathbb{V}^\sharp_{[]} \to \mathbb{V}^\sharp_{[]}$ is sound with respect to the following:

Theorem 6.6 (Soundness of $\mathtt{test}^\sharp_{\mathbb{V}^\sharp_{[]}}[\![\mathtt{B}]\!]$)

$$\forall M^\sharp \in \mathbb{V}^\sharp_{[]} : \mathtt{test}[\![\mathtt{B}]\!](\gamma_{\mathbb{V}^\sharp_{[]}}(M^\sharp)) \subseteq \gamma_{\mathbb{V}^\sharp_{[]}}(\mathtt{test}^\sharp_{\mathbb{V}^\sharp_{[]}}[\![\mathtt{B}]\!]M^\sharp)$$

Since the operator $\mathtt{test}[\![\mathtt{B}]\!]$ filters but does not add or modify the input states, $\mathtt{test}^\sharp_{\mathbb{V}^\sharp_{[]}}[\![\mathtt{B}]\!]M^\sharp \triangleq M^\sharp$ is a sound implementation for $\mathtt{test}^\sharp_{\mathbb{V}^\sharp_{[]}}[\![\mathtt{B}]\!]$. Nevertheless, we define ad-hoc functions for some useful and common cases to improve the precision of the operator. Let $M^\sharp \in \mathbb{V}^\sharp_{[]}$ such that $M^\sharp(\mathtt{x}) = [l_1, u_1]$ and $M^\sharp(\mathtt{y}) = [l_2, u_2]$.

$$\mathtt{test}^\sharp_{\mathbb{V}^\sharp_{[]}}[\![\mathtt{x} <= \mathtt{n}]\!]M^\sharp \triangleq \begin{cases} M^\sharp[\mathtt{x} \leftarrow [l_1, \min(u_1, \mathtt{n})]] & \text{if } l_1 \leq \mathtt{n} \\ \bot_e & \text{otherwise} \end{cases} \tag{6.44}$$

$$\texttt{test}^\sharp_{\mathbb{V}^\sharp_\square}[\![\texttt{x <= y}]\!]M^\sharp \triangleq \begin{cases} M^\sharp[\texttt{x} \leftarrow [l_1, \min(u_1, u_2)]][\texttt{y} \leftarrow [\max(l_1, l_2), u_2]] & \text{if } l_1 \leq u_2 \\ \bot_e & \text{otherwise} \end{cases} \tag{6.45}$$

More precise versions of the abstract test operator require bottom-up and top-down traversals of the abstract syntax tree of the boolean expression B, called the HC4 algorithm in the constraint solving community [151]. In the context of abstract interpretation, the description of these traversals can be found in [146, Section 4.6].

Abstract semantics of statements

By relying on the arithmetic and test abstract evaluation, we can finally define the *abstract interval semantics of statements*. The abstract semantics $\mathcal{S}^\sharp_{\mathbb{V}^\sharp_\square}[\![\mathtt{S}]\!] : \mathbb{V}^\sharp_\square \to \mathbb{V}^\sharp_\square$ overapproximates the set of reachable states after the execution of S, and it is sound with respect to the following.

Theorem 6.7 (Soundness of the interval abstract semantics)

$$\forall M^\sharp \in \mathbb{V}^\sharp_\square : \mathcal{S}[\![\mathtt{S}]\!](\gamma_{\mathbb{V}^\sharp_\square}(M^\sharp)) \subseteq \gamma_{\mathbb{V}^\sharp_\square}(\mathcal{S}^\sharp_{\mathbb{V}^\sharp_\square}[\![\mathtt{S}]\!]M^\sharp)$$

For each statement, the abstract evaluation starting from bottom evaluates to bottom:

$$\mathcal{S}^\sharp_{\mathbb{V}^\sharp_\square}[\![\mathtt{S}]\!]\bot_e \triangleq \bot_e \tag{6.46}$$

For `skip`, assignments and composition, the abstract semantics is defined as follows:

$$\mathcal{S}^\sharp_{\mathbb{V}^\sharp_\square}[\![\texttt{skip}]\!]M^\sharp \triangleq M^\sharp \tag{6.47}$$

$$\mathcal{S}^\sharp_{\mathbb{V}^\sharp_\square}[\![\texttt{x = A}]\!]M^\sharp \triangleq M^\sharp[\texttt{x} \leftarrow \mathcal{A}^\sharp_{\mathbb{V}^\sharp_\square}[\![\texttt{A}]\!]M^\sharp] \tag{6.48}$$

$$\mathcal{S}^\sharp_{\mathbb{V}^\sharp_\square}[\![\mathtt{S}_1\,;\,\mathtt{S}_2]\!]M^\sharp \triangleq \mathcal{S}^\sharp_{\mathbb{V}^\sharp_\square}[\![\mathtt{S}_2]\!](\mathcal{S}^\sharp_{\mathbb{V}^\sharp_\square}[\![\mathtt{S}_1]\!]M^\sharp) \tag{6.49}$$

For if statements, we first filter the states that enter each branch with the abstract test evaluation, and then we join the two resulting abstract states with the abstract join operator.

$$\mathcal{S}^\sharp_{\mathbb{V}^\sharp_\mathbb{I}}[\![\texttt{if (B) } \mathtt{S}_t \texttt{ else } \mathtt{S}_e]\!]M^\sharp \triangleq \mathcal{S}^\sharp_{\mathbb{V}^\sharp_\mathbb{I}}[\![\mathtt{S}_t]\!](\texttt{test}^\sharp_{\mathbb{V}^\sharp_\mathbb{I}}[\![\mathtt{B}]\!]M^\sharp) \cup^\sharp_{\mathbb{V}^\sharp_\mathbb{I}} \mathcal{S}^\sharp_{\mathbb{V}^\sharp_\mathbb{I}}[\![\mathtt{S}_e]\!](\texttt{test}^\sharp_{\mathbb{V}^\sharp_\mathbb{I}}[\![\neg\mathtt{B}]\!]M^\sharp) \tag{6.50}$$

While statements are the most interesting constructs, as a trivial definition for the abstract semantics might not be computable. This is due to the fact that some programs might not terminate, and simply accumulating an overapproximation of the reachable states at each iteration is not sufficient to guarantee the convergence of the analysis in finite time. To avoid nontermination, we introduce the *widening operator* which is an overapproximating operator that computes upper bounds and guarantees termination. We first give a formal definition of widening operators.

Definition 6.13 (Widening operator)
A binary operator $\nabla : \mathbb{D}^\sharp \times \mathbb{D}^\sharp \to \mathbb{D}^\sharp$ is a *widening operator* in an abstract domain $(\mathbb{D}^\sharp, \sqsubseteq^\sharp)$ if:

1. It computes upper bounds: $\forall x^\sharp_1, x^\sharp_2 \in \mathbb{D}^\sharp : x^\sharp_1 \sqsubseteq^\sharp x^\sharp_1 \nabla x^\sharp_2 \wedge x^\sharp_2 \sqsubseteq^\sharp x^\sharp_1 \nabla x^\sharp_2$
2. It enforces termination: for any infinite sequence $x^\sharp_0, x^\sharp_1, x^\sharp_2, \ldots$ in $\mathbb{D}^\sharp$, the sequence $y^\sharp_0, y^\sharp_1, y^\sharp_2, \ldots$ computed as $y^\sharp_0 \triangleq x^\sharp_0$, $y^\sharp_{i+1} \triangleq y^\sharp_i \nabla x^\sharp_{i+1}$ stabilizes after a finite time: $\exists k \geq 0 : y^\sharp_{k+1} = y^\sharp_k$.

First, we define the widening operator $\nabla_i : \mathbb{I} \times \mathbb{I} \to \mathbb{I}$ for intervals. The fundamental idea to enforce termination is to push unstable upper bounds to positive infinity and unstable lower bounds to negative infinity.

$$\bot_i \nabla_i i \triangleq i \tag{6.51}$$

$$i \nabla_i \bot_i \triangleq i \tag{6.52}$$

$$[l_1, u_1] \nabla_i [l_2, u_2] \triangleq \left[\begin{cases} l_1 & \text{if } l_1 \leq l_2 \\ -\infty & \text{otherwise} \end{cases}, \begin{cases} u_1 & \text{if } u_1 \geq u_2 \\ \infty & \text{otherwise} \end{cases} \right] \tag{6.53}$$

We can then lift the widening operator on intervals to the widening operator

$\nabla_{\mathbb{V}_{[]}^\sharp} : \mathbb{V}_{[]}^\sharp \times \mathbb{V}_{[]}^\sharp \to \mathbb{V}_{[]}^\sharp$ by applying ∇_i point-wise:

$$\bot_e \nabla_{\mathbb{V}_{[]}^\sharp} M^\sharp \triangleq M^\sharp \tag{6.54}$$

$$M^\sharp \nabla_{\mathbb{V}_{[]}^\sharp} \bot_e \triangleq M^\sharp \tag{6.55}$$

$$M_1^\sharp \nabla_{\mathbb{V}_{[]}^\sharp} M_2^\sharp \triangleq \lambda \mathrm{x} \in \mathbb{V}. M_1^\sharp(\mathrm{x}) \nabla_i M_2^\sharp(\mathrm{x}) \tag{6.56}$$

By relying on the widening operator, we can give a definition of the abstract semantics for while statements that is guaranteed to converge after a finite number of iterations.

$$\mathcal{S}^\sharp_{\mathbb{V}_{[]}^\sharp}[\![\texttt{while (B) } \mathrm{S}_b]\!] M^\sharp \triangleq \texttt{test}^\sharp_{\mathbb{V}_{[]}^\sharp}[\![\neg \mathrm{B}]\!](\lim F^\sharp) \tag{6.57}$$

$$\textbf{where } F^\sharp(M_1^\sharp) \triangleq M_1^\sharp \nabla_{\mathbb{V}_{[]}^\sharp} (M^\sharp \cup^\sharp_{\mathbb{V}_{[]}^\sharp} \mathcal{S}^\sharp_{\mathbb{V}_{[]}^\sharp}[\![\mathrm{S}_b]\!](\texttt{test}^\sharp_{\mathbb{V}_{[]}^\sharp}[\![\mathrm{B}]\!] M_1^\sharp))$$

Example 6.20 (Interval analysis with widening)
Consider the following program:

```
1 i = 0;
2 while (i < 10) {
3   i = i + 1;
4 }
```

The abstract state reached at the loop head before the first iteration is $\{\,\mathtt{i} \mapsto [0, 0]\,\}$. After one iteration, the variable `i` is incremented, so that the post state after the first iteration is $\{\,\mathtt{i} \mapsto [1, 1]\,\}$. This is merged with the first state using the $\cup^\sharp_{\mathbb{V}_{[]}^\sharp}$ operator, obtaining $\{\,\mathtt{i} \mapsto [0, 1]\,\}$. We then apply the widening operator:

$$\{\,\mathtt{i} \mapsto [0, 0]\,\} \nabla_{\mathbb{V}_{[]}^\sharp} \{\,\mathtt{i} \mapsto [0, 1]\,\} = \{\,\mathtt{i} \mapsto [0, +\infty]\,\}$$

Since the upper bound for `i` is unstable, it is widened to $+\infty$. We now reached the limit of the iterations, as by evaluating the body of the while statement again from the state $\{\,\mathtt{i} \mapsto [0, +\infty]\,\}$, we obtain the same result. We can then apply the filter $\texttt{test}^\sharp_{\mathbb{V}_{[]}^\sharp}[\![\neg(\mathtt{i} < 10)]\!]$ and observe that the final abstract state is

$\{\,\mathtt{i} \mapsto [10, +\infty]\,\}$. Observe that this is not the most precise invariant that we can infer for the variable `i`, which is `i` = 10. Advanced iteration techniques such as *decreasing iterations*, can be implemented to improve the precision of the analysis and infer this information [146, Section 4.7].

We defined a sound, computable abstract interval semantics for the WHILE language. Many of the definitions we presented rely on underlying interval operations, and as it turns out they can be used with other domains that overapproximate sets of integers. In the following sections, we give a signature of the concepts of *abstract value domain* and *relational abstract domain*. As we will observe, different domains offer different tradeoffs between precision and performance.

Non-relational abstract domains

Abstract value domains overapproximate the values of each variable individually, losing the relational information. This class of domains is widely used in real-world scenarios because they are typically computationally efficient. The interval abstract domain presented in the previous section belongs to this category.

Definition 6.14 (Abstract value domain)
A poset $(\mathbb{E}, \sqsubseteq_e)$ is an *abstract value domain* when it exposes:

- A monotonic concretization function $\gamma_e : \mathbb{E} \to \wp(\mathbb{Z})$
- A top element $\top_e$
- A bottom element $\bot_e$
- Sound abstraction of constants, written $\mathtt{n}_e$ for $\mathtt{n} \in \mathbb{Z}$
- Sound abstractions of the arithmetic operators $+_e, -_e, \times_e, /_e$
- Sound abstractions of set union and intersection $\cup_e, \cap_e$
- A widening operator ∇_e
- Optionally, a Galois Connection $\mathbb{E} \underset{\alpha_e}{\overset{\gamma_e}{\leftrightarrows}} \wp(\mathbb{Z})$

Furthermore, the elements of the domain must be *finitely representable*, and the abstractions of constants, arithmetic operators, and set union and intersection must be *computable*.

By using the abstractions of a value domain $\mathbb{E}$ in the definitions presented in the previous section, it is possible to automatically lift the arithmetic abstract evaluation and the abstract semantics to $\mathbb{E}$. We denote the resulting abstract domain as $\mathbb{V}^\sharp_\mathbb{E}$. If the abstract value domain enjoys a Galois Connection $\mathbb{E} \underset{\alpha_e}{\overset{\gamma_e}{\leftrightarrows}} \wp(\mathbb{Z})$, then the connection is lifted to $\mathbb{V}^\sharp_\mathbb{E} \underset{\alpha_{\mathbb{V}^\sharp_\mathbb{E}}}{\overset{\gamma_{\mathbb{V}^\sharp_\mathbb{E}}}{\leftrightarrows}} \mathbb{D}$. There exists a large library of abstract value domains, and here we report some of them:

Signs [26]. The sign domain infers constraints about the sign of each individual variable. It is computationally inexpensive, as each variable is simply associated to its sign (positive, negative, zero, or any). This domain is not considered useful in real-world scenarios, as it fails to provide sufficiently precise information. Nevertheless, the sign domain is often used as an introductory example of an abstract domain for teaching purposes.

Powerset [146]. The powerset domain associates to each variable a finite set of possible values, namely $\mathtt{x} \in \{n_1, n_2, \ldots, n_k\}$ with $|\{n_1, n_2, \ldots, n_k\}| < \infty$. The domain is precise, but if the maximum size of the powerset is set too large, it can quickly become computationally expensive.

Excluded powerset [152]. An interesting variant of the regular powerset is the *excluded powerset*, capable of representing not only the elements that might occur but also the ones that *definitely cannot occur*. The constraints have type $\mathtt{x} \in \{n_1, n_2, \ldots, n_k\}$ or $\mathtt{x} \notin \{n_1, n_2, \ldots n_k\}$ with $k < \infty$. This domain proved itself useful in software verification competitions to improve the precision of the C analysis [152]. The reason is that often C functions return integer codes such that any non-zero number represents an error. It is therefore important to be able to precisely represent possibly infinite non-convex sets.

Intervals [26]. The interval domain, studied in the previous section, infers constraints of type $\mathtt{x} \in [l, u]$. The domain is widely used in practice, as it offers good precision with a low performance cost.

Disjunctive intervals domain [153]. The disjunctive intervals domain generalizes the interval domain by assigning each variable to a finite disjunction of intervals, namely $\mathtt{x} \in \cup_{i=1}^{n} [l_i, u_i]$. For this reason, this domain also subsumes both the powerset and the excluded powerset domains.

Congruences [154, 155]. The congruence domain represents constraints of type $\mathtt{x} \equiv n \bmod m$. The domain is useful for variables used as array indices, which are often congruent to the size of the array elements. Another common use is verifying that variables used as pointer offsets are correctly aligned.

Relational abstract domains

While value domains abstract each variable individually, it is possible to enhance the precision of the analysis by designing domains that do not lose relational information. First, we give the most general signature of an abstract domain.

Definition 6.15 (Abstract domain)
A poset $(\mathbb{D}^\sharp, \sqsubseteq^\sharp)$ is an *abstract domain* when it exposes:

- A monotonic concretization function $\gamma : \mathbb{D}^\sharp \to \mathbb{D}$
- A top element $\top^\sharp$ that represents $\mathbb{S}$
- A bottom element $\bot^\sharp$ that represents $\emptyset$
- Sound abstractions of arithmetic binary operations in conditionals $\mathtt{test}^\sharp[\![\mathtt{A}_1 \diamond \mathtt{A}_2]\!]$ (for $\diamond \in \{\mathtt{<}, \mathtt{<=}, \mathtt{>}, \mathtt{>=}, \mathtt{==}, \mathtt{!=}\}$)
- Sound abstraction of assignments $\mathbb{S}^\sharp[\![\mathtt{x = A}]\!]$
- Sound abstractions of set union and intersection $\cup^\sharp, \cap^\sharp$
- A widening operator ∇
- Optionally, a Galois Connection $\mathbb{D} \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \mathbb{D}^\sharp$

Furthermore, the elements of the domain must be *finitely representable*, and the abstractions of tests, assignments, and set union and intersection must be *computable*.

If $\mathbb{D}^\sharp$ is an abstract domain, then the following defines a sound, computable abstract semantics for the WHILE language:

$$\texttt{test}^\sharp[\![\texttt{tt}]\!]M^\sharp \triangleq M^\sharp \tag{6.58}$$

$$\texttt{test}^\sharp[\![\texttt{ff}]\!]M^\sharp \triangleq \bot^\sharp \tag{6.59}$$

$$\texttt{test}^\sharp[\![\texttt{B}_1 \texttt{ \&\& } \texttt{B}_2]\!]M^\sharp \triangleq \texttt{test}^\sharp[\![\texttt{B}_1]\!]M^\sharp \cap^\sharp \texttt{test}^\sharp[\![\texttt{B}_2]\!]M^\sharp \tag{6.60}$$

$$\texttt{test}^\sharp[\![\texttt{B}_1 \texttt{ || } \texttt{B}_2]\!]M^\sharp \triangleq \texttt{test}^\sharp[\![\texttt{B}_1]\!]M^\sharp \cup^\sharp \texttt{test}^\sharp[\![\texttt{B}_2]\!]M^\sharp \tag{6.61}$$

$$\mathcal{S}^\sharp[\![\texttt{skip}]\!]M^\sharp \triangleq M^\sharp \tag{6.62}$$

$$\mathcal{S}^\sharp[\![\texttt{S}_1\texttt{ ; }\texttt{S}_2]\!]M^\sharp \triangleq \mathcal{S}^\sharp[\![\texttt{S}_2]\!](\mathcal{S}^\sharp[\![\texttt{S}_1]\!]M^\sharp) \tag{6.63}$$

$$\mathcal{S}^\sharp[\![\texttt{if (B) } \texttt{S}_t \texttt{ else } \texttt{S}_e]\!]M^\sharp \triangleq \mathcal{S}^\sharp[\![\texttt{S}_t]\!](\texttt{test}^\sharp[\![\texttt{B}]\!]M^\sharp) \cup^\sharp \mathcal{S}^\sharp[\![\texttt{S}_e]\!](\texttt{test}^\sharp[\![\neg\texttt{B}]\!]M^\sharp) \tag{6.64}$$

$$\mathcal{S}^\sharp[\![\texttt{while (B) } \texttt{S}_b]\!]M^\sharp \triangleq \texttt{test}^\sharp[\![\neg\texttt{B}]\!](\operatorname{lim} F^\sharp) \tag{6.65}$$

$$\textbf{where } F^\sharp(M_1^\sharp) \triangleq M_1^\sharp \nabla (M^\sharp \cup^\sharp \mathcal{S}^\sharp[\![\texttt{S}_b]\!](\texttt{test}^\sharp[\![\texttt{B}]\!]M_1^\sharp))$$

Remark 6.7 (Negation in conditionals)
Observe that we do not require abstract domains to provide $\texttt{test}^\sharp[\![\neg\texttt{B}]\!]$ nor do we define it in the abstract semantics. This is because it is always possible to syntactically remove all negations in conditional expressions by using DeMogan's laws. To simplify, we assume that all negations have been syntactically removed.

Remark 6.8 (Abstract value domains and abstract domains)
For any abstract value domain $\mathbb{E}$, $\mathbb{V}^\sharp_{\mathbb{E}}$ is an abstract domain.

Now that we formally defined what an abstract domain is, we present some *relational* domains. Compared to value domains, they have the ability to infer relationships between variables.

Linear Equalities [156]. Also known as *Karr's domain*, the linear equalities domain is able to infer affine relationships among variables. The invariants have the form $\bigwedge_{j=1}^{m} \sum_{i=1}^{|\mathbb{V}|} (k_{ij} \cdot \mathtt{x}_i) = n_j$.

Polyhedra [157]. The polyhedra abstract domain infers linear inequalities among variables. It is used when superior precision is required, as the domain yields

TABLE 6.1. List of some existing numeric abstract domains

Domain	Type	Constraints
Signs [26]	Value domain	$\mathtt{x} \in \pm$
Powerset [146]	Value domain	$\mathtt{x} \in \{n_1, n_2, \ldots, n_k\}$
Excluded powerset [152]	Value domain	$\mathtt{x} \in \{n_1, n_2, \ldots, n_k\} \vee \mathtt{x} \notin \{n_1, n_2, \ldots, n_k\}$
Intervals [26]	Value domain	$\mathtt{x} \in [l, u]$
Disjunctive intervals [153]	Value domain	$\mathtt{x} \in \cup_{i=1}^{n} [l_i, u_i]$
Congruences [154, 155]	Value domain	$\mathtt{x} \equiv n \bmod m$
Linear equalities [156]	Relational domain	$\sum_{i=1}^{\lvert\mathbb{V}\rvert} (k_i \cdot \mathtt{x}_i) = n$
Polyhedra [157]	Relational domain	$\sum_{i=1}^{\lvert\mathbb{V}\rvert} (k_i \cdot \mathtt{x}_i) \leq n$
Octagons [158]	Relational domain	$\pm\mathtt{x} \pm \mathtt{y} \leq n$

precise numeric invariants. Nevertheless, the domain is computationally expensive, and the analysis may be time-consuming.

Octagons [158]. The octagon abstract domain strikes a balance between intervals and polyhedra, and it can infer relational information of the form $\pm\mathtt{x} \pm \mathtt{y} \leq n$. While it is more precise than intervals, it is computationally more efficient than polyhedra.

Remark 6.9 (Coefficients in relational domains)
Note that for relational domains, we generally have to assume that the coefficients are rationals, even if we intend to represent sets of integer-valued environments. The reason is that using integers as coefficients would lead to unsound algorithms [146, Chapter 5].

Example 6.21 (Relational analysis)
Consider the following program:

```
y = x; z = x-y
```

If we run an interval analysis on the program and we consider as initial value for x the interval [0, 10], then at the end we find that the value of z is [–10, 10]. This is because the interval abstract domain is not precise enough to

represent the constraint $x = y$. By using any of the relational abstract domains presented in this section, it is possible to infer that at the end of the program z is 0.

Remark 6.10 (Non-numeric abstract domains)
We focused our attention on *numeric* domains, namely abstract domains that overapproximate the values of numeric variables. There exists a rich library of domains that are not numeric and abstract other aspects of programs such as memory blocks [159, 160], dynamic memory allocation [161, 162], arrays [163, 164, 165, 166], dynamic types [162, 167, 168], class invariants [169], lists [170, 171, 172], trees [170, 172, 173], and string contents [174, 175, 176, 177, 178, 179, 180, 181].

Combining domains: reduced products

Different domains can represent complementary information about a variable. For instance, the congruence domain can infer that a variable is odd, while the interval domain gives an upper and lower bound for the values of the variable. Combining the two can result in a strictly more precise analysis. Consider, for instance, the following program (taken from [146]):

```
1 x = 1
2 while (x <= 10) {
3   x = x + 2
4 }
```

By applying the *decreasing iterations* technique described in [146, Section 4.7], the interval domain can infer that $x \in [11, 12]$ after exiting the loop, while the congruence domain infers that x is odd. If we combine this information, we can observe that x is exactly 11, which would not be possible by using either of the domains alone.

A *reduced product* is a domain that is defined as a pair of underlying abstract domains equipped with a reduction operator to improve the precision of the two. The concretization function of the product is the intersection of the concretization of the individual domains.

Definition 6.16 (Reduced product)
Let $\mathbb{D}_1^\sharp, \mathbb{D}_2^\sharp$ be two abstract domains, and let $\rho : \mathbb{D}_{1\times 2}^\sharp \to \mathbb{D}_{1\times 2}^\sharp$ be a given *reduction operator* that refines $\mathbb{D}_1^\sharp$ and $\mathbb{D}_2^\sharp$. The *reduced product* $\mathbb{D}_{1\times 2}^\sharp$ is an abstract domain composed of the following:

- $\mathbb{D}_{1\times 2}^\sharp \triangleq \mathbb{D}_1^\sharp \times \mathbb{D}_2^\sharp$
- Partial order $(M_1^\sharp, M_2^\sharp) \sqsubseteq_{1\times 2}^\sharp (N_1^\sharp, N_2^\sharp) \iff^{\triangle} M_1^\sharp \sqsubseteq_1^\sharp N_1^\sharp \wedge M_2^\sharp \sqsubseteq_2^\sharp N_2^\sharp$
- Monotonic concretization function $\gamma_{1\times 2}(M_1^\sharp, M_2^\sharp) \triangleq \gamma_1(M_1^\sharp) \cap \gamma_2(M_2^\sharp)$
- Top element $\top_{1\times 2}^\sharp \triangleq (\top_1^\sharp, \top_2^\sharp)$
- Bottom element $\bot_{1\times 2}^\sharp \triangleq (\bot_1^\sharp, \bot_2^\sharp)$
- Sound operator for tests:

$$\texttt{test}_{1\times 2}^\sharp[\![\mathtt{B}]\!](M_1^\sharp, M_2^\sharp) \triangleq \rho(\texttt{test}_1^\sharp[\![\mathtt{B}]\!]M_1^\sharp, \texttt{test}_2^\sharp[\![\mathtt{B}]\!]M_2^\sharp)$$

- Sound operator for abstract execution:

$$\mathbb{S}_{1\times 2}^\sharp[\![\mathtt{S}]\!](M_1^\sharp, M_2^\sharp) \triangleq \rho(\mathbb{S}_1^\sharp[\![\mathtt{S}]\!]M_1^\sharp, \mathbb{S}_2^\sharp[\![\mathtt{S}]\!]M_2^\sharp)$$

- Sound operators for set union and intersection:

$$(M_1^\sharp, M_2^\sharp) \cup^\sharp (N_1^\sharp, N_2^\sharp) \triangleq \rho(M_1^\sharp \cup_1^\sharp N_1^\sharp, M_2^\sharp \cup_2^\sharp N_2^\sharp)$$
$$(M_1^\sharp, M_2^\sharp) \cap^\sharp (N_1^\sharp, N_2^\sharp) \triangleq \rho(M_1^\sharp \cap_1^\sharp N_1^\sharp, M_2^\sharp \cap_2^\sharp N_2^\sharp)$$

- Widening operator $(M_1^\sharp, M_2^\sharp)\nabla_{1\times 2}(N_1^\sharp, N_2^\sharp) \triangleq (M_1^\sharp \nabla_1 N_1^\sharp, M_2^\sharp \nabla_2 N_2^\sharp)$
- Abstraction function $\alpha_{1\times 2}(M_1^\sharp, M_2^\sharp) \triangleq (\alpha_1(M_1^\sharp), \alpha_2(M_2^\sharp))$ if α_1 and α_2 exist

The reduction operator ρ must respect the following condition:

$$\begin{aligned}(N_1^\sharp, N_2^\sharp) = \rho(M_1^\sharp, M_2^\sharp) \implies\ & \gamma_{1\times 2}(N_1^\sharp, N_2^\sharp) = \gamma_{1\times 2}(M_1^\sharp, M_2^\sharp) \wedge \\ & \gamma_1(N_1^\sharp) \subseteq \gamma_1(M_1^\sharp) \wedge \\ & \gamma_2(N_2^\sharp) \subseteq \gamma_2(M_2^\sharp)\end{aligned}$$

The first condition states the soundness, while the two inclusions state that each element is strengthened in its respective domain. As an additional condition, the reduction operator must be *computable*. If $\mathbb{D}_1^\sharp$ and $\mathbb{D}_2^\sharp$ are abstract value domains, the reduction can be defined directly on the abstract values of each variable.

Remark 6.11 (Optimal reduction)
In case the two domains in a reduced product enjoy a Galois connection, the optimal reduction corresponds to the following:

$$\rho(M_1^\sharp, M_2^\sharp) \triangleq (\alpha_1(\gamma_{1\times 2}(M_1^\sharp, M_2^\sharp)), \alpha_2(\gamma_{1\times 2}(M_1^\sharp, M_2^\sharp)))$$

This reduction is *optimal,* namely it is the most precise sound reduction operator for the two domains.

Remark 6.12 (Widening in reduced product domains)
Observe that in Def. 6.16 we apply the reduction operator ρ after joins and intersections, but not after the widening operator. The reason is that if we apply ρ after the widening, termination is no longer guaranteed [146, Section 6.2]. For instance, the widening operator for intervals enforces convergence by setting interval bounds to infinity. If those bounds are then refined back to finite numbers by the reduction, the analysis might not terminate.

Example 6.22 (Interval-congruence reduced product)
Let $\mathbb{C}$ be the congruence value domain described in the previous section. We define a reduction $\rho_{\mathbb{I}\times\mathbb{C}} : \mathbb{I} \times \mathbb{C} \to \mathbb{I} \times \mathbb{C}$ between intervals and congruences. The reduction operates directly on the abstract values:

$$\rho_{\mathbb{I}\times\mathbb{C}}([l, u], a\mathbb{Z} + b) \triangleq \begin{cases} (\bot_i, \bot_c) & \text{if } l' > u' \\ ([l', l'], 0\mathbb{Z} + l') & \text{if } l' = u' \\ ([l', u'], a\mathbb{Z} + b) & \text{if } l' < u' \end{cases}$$
$$\textbf{where } l' = \min\{\, n \mid n \geq l \wedge n \equiv b \bmod a \,\}$$

$$u' = \max\{\, n \mid n \leq u \wedge n \equiv b \bmod a \,\}$$

Remark 6.13 (Direct product domain)
The reduction operator can be trivially implemented as follows:

$$\rho(M_1^\sharp, M_2^\sharp) \triangleq (M_1^\sharp, M_2^\sharp)$$

This reduction does not enhance the precision of the analysis, as it does not leverage the information available in the two underlying domains. The domain where the reduction is trivial is known as the *direct product domain* [146, Section 6.2].

6.4.4. Static analysis tools based on abstract interpretation

During the years, many static analyzers based on abstract interpretation theory have been developed.

ASTRÉE [27, 182, 183, 184]. The ASTRÉE analyzer is a commercial static analysis tool specifically designed to verify the correctness of large embedded safety-critical software written in C. The analyzer was able to prove the absence of runtime errors in the flight control codes of the Airbus fly-by-wire systems.

CLOUSOT [185]. CLOUSOT can verify contract specifications of individual functions. The tool checks every method in isolation using an assume-guarantee reasoning: first, it assumes the precondition, and then it asserts the postcondition. At its core, CLOUSOT is an abstract interpreter that infers program facts. In addition to verifying contracts, CLOUSOT can also report regular runtime errors, such as null-pointer dereferences.

FRAMA-C [137, 186, 187] FRAMA-C is a platform for static analysis, dynamic analysis, and deductive verification. It supports a plugin system, which makes it possible to extend the existing analyses and add new ones. The EVA plugin [186, 187] infers numeric invariants for C code by relying on abstract interpretation theory.

IKOS [188]. NASA developed the IKOS open-source static analysis platform. The tool has a frontend for C and C++ programs based on LLVM [189], and it is optimized for the verification of real-time software. The SEAHORN automated verifier [190] relies on IKOS to infer numeric invariants for LLVM-based languages.

INFER [191, 192]. Meta developed and maintains INFER, which is a multi-language static analyzer for C, C++, Java, and Objective-C. The tool is used to analyze large real-world code bases during the continuous integration process. INFER's main objective is to be fast and report a low number of false positives, so that soundness is sometimes traded for performance.

JULIA [193, 194]. JULIA is a static analyzer based on abstract interpretation for Java. It can analyze Java bytecode, which makes it possible to easily support other JVM-based languages such as Scala. Furthermore, Julia supports a security analysis to detect injection attacks [194].

LISA [195, 196, 197]. LISA is an analyzer specifically designed for teaching. While many tools based on abstract interpretation often demand a high level of expertise before being proficient, LISA has been conceived to be simple to understand, use, and extend.

MOPSA [48, 198, 199, 167]. In this work, we implement our algorithms within the MOPSA framework, a sound static analysis platform to develop abstract interpretation-based analyses. MOPSA has a modular architecture, which makes it easy to extend and combine existing abstract domains. The tool supports both C [198] and Python [167], and can combine the two in a multi-language analysis [199]. MOPSA offers a large set of ready-to-use abstract numeric domains provided by the APRON [200] library.

PYSA [201]. PYSA is a static analyzer based on abstract interpretation for *taint analysis* [202] of Python programs. It is used at Meta to analyze real-world large Python code bases and prove the absence of injection attacks.

TAJS [168, 203, 204, 205, 206]. TAJS is a sound static analyzer for Javascript programs that reports type-related errors. It relies on a simplified underlying

language to which Javascript code is compiled to, and this makes it easier to support a large set of high-level complex constructs.

VERASCO [207, 208, 209, 210]. The VERASCO abstract interpreter can analyze C programs, and its soundness is formally proved with the COQ [129] proof assistant. VERASCO is integrated within the formally-verified COMPCERT [211] C compiler so that not only the soundness of the analysis results is mathematically guaranteed, but also these guarantees transfer to the compiled program.

6.5. Conclusion

In this section, we summarized the principles of static analysis by abstract interpretation. First, we specified the concrete semantics of the WHILE language, i.e. a precise mathematical formulation of programs behaviour. Then, we defined different classes of program properties, which are represented as sets of elements that have the property. Finally, we described a sound, computable abstract semantics for programs that is parametric on the underlying abstract domain used to infer numeric invariants.

Chapter 7

Sound Abstract Safety Nonexploitability Analysis

Runtime errors that can be triggered by an attacker are sensibly more dangerous than others, as they not only result in program failure, but can also be exploited and lead to security breaches such as Denial-of-Service attacks or remote code execution. In this chapter, we introduce a novel analysis based on abstract interpretation in order to prove that a program is *safety-nonexploitable*, namely it does not present runtime errors that can be triggered by an attacker.

In Section 7.1 and Section 7.2 we introduce the problem we tackle and we motivate the need for our work, while in Section 7.3 we introduce the basic ideas of a well-known technique called *taint analysis* that we use later in this chapter. Section 7.4 and Section 7.5 present respectively the syntax and the semantics of the language that we consider, which supports features such as *nondeterminism* and *runtime user input reads*. In Section 7.6 we formally define *safety-nonexploitability* as a hyperproperty, and in Section 7.7 we introduce a concrete semantics that precisely captures the set of user-controlled variables, which is useful to prove that a program is safety-nonexploitable. Section 7.8 finally presents the computable abstract semantics, i.e. the analysis, which can prove a program to be safety-nonexploitable. The proofs of the theoretical results are reported in Appendix A. This chapter and the following are based on a work published at VMCAI 2024 [53].

7.1. Introduction

Program failures that can be triggered by a malicious user are sensibly more dangerous than others, as they can lead to security breaches. Attackers can exploit well-known runtime errors, such as index out-of-bounds and double free, to perform dangerous attacks including Denial-of-Service (DoS) attacks or remote code execution. Numerous companies identified such exploitable vulnerabilities in their systems, including Meta [45], Apple [46], and Google [47]. Microsoft recently published a report showing that consistently over 20 years, around 70% of the security breaches that have been reported in their systems are due to exploitable memory corruption [29]. As it is difficult to identify program errors with manual inspection, static analysis is an invaluable tool to automatically detect them.

While sound static analyzers can report *all* possible runtime errors, including the exploitable ones, they often raise a high number of false positives. If the noise generated by the false alarms is elevated, the report of the analyzer quickly becomes unintelligible, and it is then difficult to identify the true exploitable runtime errors. In order to filter out the warnings that do not concern security issues, it is necessary to combine a traditional analysis for safety properties with a security analysis for hyperproperties [50].

We bridge the gap between classic safety and security. We first formalize *safety-nonexploitability* as a 2-hypersafety property, and then we propose an alternative characterization based on *semantically tainted* (i.e. user-controlled) variables. We leverage such a characterization to put forward a sound analysis by abstract interpretation [26] that can prove the absence of exploitable runtime errors. Our analysis has the capability to classify each warning by its threat level (security-related or not), which makes the report of the analyzer more intelligible.

We leverage an underlying abstract numeric domain to infer numeric invariants, which we pair with a *semantic taint analysis* that tracks the set of user-controlled variables. Combining the two is necessary in order to infer an overapproximation of the exploitable runtime errors. By taking advantage of the semantic information inferred by the abstract numeric domain, our taint analysis achieves enhanced precision compared to traditional methods. Furthermore, our framework can handle programming language features that are essential to analyze real-world programs, such as nondeterminism and runtime user input. While we formalize

```
1  #include <stdio.h>
2  #include <string.h>
3
4  void use_input(const char* input) {
5    char dest[10];
6    strcpy(dest, input);
7  }
8
9  void main() {
10   char buff[100];
11   fgets(buff, sizeof(buff), stdin);
12   use_input(buff);
13 }
```

FIGURE 7.1. C program with exploitable buffer overflow

our theoretical framework on a simple toy language, our implementation supports almost the full C specification. As discussed in Chapter 8, our analyzer can prove the absence of a wide variety of exploitable runtime errors, including buffer overflows and invalid memory accesses.

In this work, we focus on the exploitability of safety errors, such as null pointer deferences and array out-of-bounds accesses. Nevertheless, other interesting types of vulnerabilities could be exploited by an attacker. For instance, in *algorithmic complexity attacks* [35] a malicious user exploits the fact that an application has poor performance characteristics under certain inputs. These types of vulnerabilities include Regular Expression Denial of Service (ReDoS) attacks, discussed in Part II of this manuscript. Another example of software error that could be triggered by an attacker is non-termination: if an application enters an infinite loop due to a user action, then such a software system should be considered non-termination-exploitable. Our framework does not currently support these different types of errors, and in future work we would like to extend our analysis to handle them.

7.2. Motivation

Figure 7.1 represents an exploitable program where an external user can input a sequence up to 100 characters long. At line 6, the program attempts to copy the buffer's contents into another array. However, due to the second array's smaller size, if the user inputs more than 10 characters including the terminating `'\0'`,

a buffer overflow occurs. A malicious user can take advantage of these types of vulnerabilities to execute sophisticated, dangerous attacks. For instance, the famous Code Red computer worm exploited inadequate array bounds checking in Microsoft's IIS web server [41]. By providing a carefully crafted input string to the application, the virus was able to perform system-level arbitrary code execution. There are numerous other examples of well-known attacks that exploit runtime errors: among them we find the Morris Worm (1988) [42], SQL Slammer (2003) [43], and Heartbleed (2014) [44]. Memory-related runtime errors regularly appear among the official Top 25 Most Dangerous Software Weaknesses list yearly published by MITRE, and from 2021 to 2023 out-of-bounds write has been ranked as *the* most dangerous vulnerability [212].

While techniques such as testing and human inspection by security experts are useful to detect (exploitable) runtime errors, the only option to rule out their existence is through formal methods. In particular, abstract interpretation has been effective in proving the absence of program failures in real-time avionics software [183]. While analyses by abstract interpretation are *sound*, they can often raise false positives. If the noise generated by the false alarms is too high, the analyzer quickly becomes unusable.

Reducing the number of false alarms is usually achieved by employing more precise abstract domains. Our work takes an orthogonal approach to the problem, reducing the number of alarms by reporting the subset of possible runtime errors that can be triggered by an attacker. As these errors are comparatively more dangerous than the others, the report of the analyzer becomes more intelligible, enhancing the usefulness of sound static analysis tools. Girol et al. make the same observation, leveraging the concept of *robust reachability* to identify errors that are relatively more dangerous [213]. A bug is *robustly reachable* if there exists a user input for which the bug is always reached, regardless of the value of the uncontrolled input. The main difference with our concept of exploitability, is that we require the user input to be actually used in triggering the bug, while strong reachability does not (see Section 7.9 for a detailed comparison).

Taint analysis, informally introduced in the following section, is a popular technique used in computer security to track the flow of untrusted data within a program, and we leverage this method to prove safety-nonexploitability. While

taint analyzers often rely on heuristics to track the flow of unsafe data [127], our approach is *semantic,* namely grounded in a definition based on the formal semantics of programs. While formal methods techniques extensively studied the verification of security properties such as *noninterference* [123, 124, 125] (see Section 6.3), the nonexploitability analysis is, to the best of our knowledge, uninvestigated (see Section 7.9 for a comparison). This work bridges the gap between classic safety properties analysis and security hyperproperties [50] in order to rule out the existence of exploitable runtime errors in a software system.

7.3. Taint analysis

In this section, we informally introduce the main ideas of a technique called *taint analysis,* which tracks the set of variables that are user-controlled in a program. This type of analysis is useful in security applications to find dangerous flows of data from untrusted user input (called *sources*) to sensible program locations (called *sinks*). Taint analysis remains the most applied technique in static analysis of Android apps [202], and many analyzers for real-world applications rely on it [127, 194, 201, 214]. We introduce this technique because in this work we rely on a semantic taint analysis in order to prove that a program is safety-nonexploitable.

The taint analysis $\texttt{tainted}[\![\texttt{S}]\!] : \wp(\mathbb{V}) \to \wp(\mathbb{V})$ inductively collects the set of possibly user-controlled variables after the execution of statement `S`, assuming a set of previously tainted variables T. The primary way taint is propagated is through assignments. Consider for instance `x = y`, and assume that `y` is tainted. Then, `x` also becomes tainted, as the user can influence the value of `x`. This type of taint propagation is known as an *explicit flow* (see Example 6.8), and it is captured by the following rule:

$$\texttt{tainted}[\![\texttt{x = A}]\!]T \triangleq \{\,\texttt{y} \in T \mid \texttt{y} \neq \texttt{x}\,\} \cup \{\,\texttt{x} \mid \exists \texttt{y} \in \texttt{vars}[\![\texttt{A}]\!] : \texttt{y} \in T\,\} \tag{7.1}$$

Where $\texttt{vars}[\![\texttt{A}]\!]$ is the set of variables that syntactically appear in `A`. Another type of taint propagation is through *implicit flows* (see Example 6.9), where the values of the variables are influenced by the program control. Consider for instance the program `if (x==0) y = 1 else y = 2`. If `x` is tainted, the user can possibly influence whether `y` is 1 or 2, so that `y` should be tainted. The set of tainted variables after the

execution of an if statement then corresponds to: 1) the variables tainted in the then branch; 2) the variables tainted in the else branch; 3) if the condition is tainted, the variables that are syntactically assigned inside of the branches, denoted as $\texttt{assigned}[\![\texttt{S}_t]\!]$ and $\texttt{assigned}[\![\texttt{S}_e]\!]$. The following rule detects implicit flows:

$$\begin{aligned}\texttt{tainted}[\![\texttt{if (B) S}_t\ \texttt{else S}_e]\!]T \triangleq & \\ & \texttt{tainted}[\![\texttt{S}_t]\!]T \cup \texttt{tainted}[\![\texttt{S}_e]\!]T \\ \cup & \begin{cases} \texttt{assigned}[\![\texttt{S}_t]\!] \cup \texttt{assigned}[\![\texttt{S}_e]\!] & \text{if } \exists \texttt{y} \in \texttt{vars}[\![\texttt{B}]\!] : \texttt{y} \in T \\ \emptyset & \text{otherwise} \end{cases}\end{aligned} \tag{7.2}$$

Since implicit flows are generally considered less dangerous than explicit flows, most taint analyzers simply ignore them and propagate taint only through assignments [127, 194, 201, 214]. This results in fewer tainted variables, but could also miss some significant flows of untrusted data that are difficult to observe with manual inspection. In our work, we do take implicit flows into account.

The analysis that we present in this section is sound in the sense that it returns an overapproximation of the user-controlled variables after the execution of a statement. Nevertheless, the analysis is purely syntactic, and it ignores the semantics of the program. It is possible, by taking the values of the variables into account, to improve the precision of the analysis. Consider for instance the program `x = y-y`, and assume that `y` is tainted. Then, the syntactic analysis would infer that `x` is tainted as well, because it is assigned to an expression that contains a tainted variable. Nevertheless, the value of `x` after the execution of the assignment is always zero, namely the user is not able to control its value. In Section 7.8, we put forward a *semantic* taint analysis that takes the values of the variables into account to achieve enhanced precision.

7.4. Syntax

The WHILE language that we introduced in Section 6.1 does not support significant features such as *nondeterminism*, which is the capability to read a random integer at runtime. Nondeterminism is necessary in order to model the environment in which a program is executed. Another important feature that is relevant in a

```
P := S                                              (Programs)
S := skip                                           (Statements)
   | x = input()
   | x = rand()
   | x = A
   | S; S
   | if (B) S else S
   | while (B) S
A := n ∈ ℤ                                          (Arithmetic Expressions)
   | x ∈ 𝕍
   | A ◇ A  (◇ ∈ {+, -, *, /})
B := tt                                             (Boolean Expressions)
   | ff
   | ¬B
   | B ◇ B  (◇ ∈ {&&, ||})
   | A ◇ A  (◇ ∈ {<, <=, >, >=, ==, !=})
```

FIGURE 7.2. Syntax of the WHILE language with nondeterminism

security context is *dynamic input reads*, that is the capability to read a value from the user at runtime. In Figure 7.2 we present the syntax of the WHILE language modified to support nondeterminism and dynamic user input reads. Expressions are deterministic, and nondeterminism is isolated in the language in specific statements (`rand`, `input`) to simplify the presentation. The fundamental difference between `input` and `rand` is that data read from the former is controlled by the user, and therefore relevant from a security point of view.

7.5. Semantics

In this section, we not only extend the concrete semantics of the WHILE language to support nondeterminism and dynamic user input, but we also put forward a reachability semantics that associates input states with output states. Furthermore, the semantics inductively collects runtime errors, which is necessary in order to

express safety-nonexploitability.

The *program states* are triplets $(m, i, r) \in \mathbb{M} \times \mathbb{Z}^\infty \times \mathbb{Z}^\infty \triangleq \mathbb{S}$. The first element is the program memory, the second is the unbounded sequence of inputs provided by the user, and the third is the unbounded sequence of random numbers. As we discuss later in this section, by embedding the sequence of nondeterministic choices into the states, we have the advantage of supporting nondeterministic constructs with a deterministic semantics (i.e., to each input state, we associate at most one output state). Observe that for the sake of conciseness and to avoid heavy mathematical notation, in this chapter we use the same symbols introduced in Chapter 6 for the states, domains, and concrete semantics.

We explicitly represent states in which a runtime error occurred by setting a special *return* variable to 1. All error-free states have the return variable, denoted as `ret`, set to 0. Programs cannot read nor write explicitly to `ret`, as it cannot syntactically appear in statements. Our semantics relies on pairs of initial-reachable states. A set of initial-reachable states is a relation $R \in \wp(\mathbb{S} \times \mathbb{S}) \triangleq \mathbb{D}$.

In this section, we will define the *reachability semantics* of statements $\mathcal{S}[\![\mathtt{S}]\!] : (\mathbb{D} \times \mathbb{D}) \to (\mathbb{D} \times \mathbb{D})$ by induction. The first element in the input pair is the set of pre-post states that reach the current statement without encountering an error, while the second is the set of pre-post states that previously resulted in an error. $\mathcal{S}[\![\mathtt{S}]\!](R, E)$ outputs both the reachable and the error states after executing `S`. The set of initial states is defined as follows.

$$I \triangleq \{\, ((m, i, r), (m, i, r)) \mid (m, i, r) \in \mathbb{S}, m(\mathtt{ret}) = 0 \,\} \tag{7.3}$$

We define the semantics of programs $\mathcal{S}[\![\mathtt{P} := \mathtt{S}]\!] \in \mathbb{D}$ by merging the reachable states at the end of the program with those that resulted in an error.

$$\mathcal{S}[\![\mathtt{P} := \mathtt{S}]\!] \triangleq \textbf{let } (R, E) = \mathcal{S}[\![\mathtt{S}]\!](I, \emptyset) \textbf{ in } R \cup E \tag{7.4}$$

We now define by structural induction the reachability semantics of statements. As usual, the `skip` statement does not modify the input.

$$\mathcal{S}[\![\mathtt{skip}]\!](R, E) \triangleq (R, E) \tag{7.5}$$

For the input read statement, we update the memory by assigning the first

number in the infinite input sequence to the assigned variable, and then we shift the input sequence. The operators `hd` and `tl` respectively extract the head and the tail of a sequence.

$$\mathcal{S}[\![\texttt{x = input()}]\!](R, E) \triangleq (R \circ \{\, ((m, i, r), (m[\texttt{x} \leftarrow \texttt{hd}(i)], \texttt{tl}(i), r) \mid (m, i, r) \in \mathbb{S}) \,\}, E) \tag{7.6}$$

The random read statement is similar to the input read statement, but uses the infinite sequence of random numbers.

$$\mathcal{S}[\![\texttt{x = rand()}]\!](R, E) \triangleq (R \circ \{\, ((m, i, r), (m[\texttt{x} \leftarrow \texttt{hd}(r)], i, \texttt{tl}(r)) \mid (m, i, r) \in \mathbb{S}) \,\}, E) \tag{7.7}$$

Assignments can result in errors, which in our semantics are represented as states where `ret` is 1. If a runtime error occurs, the program sets `ret` to 1 and adds the state to the second element of the output pair. Error states are collected throughout the execution, and are propagated at the end of the program even in case of non-termination. Note, however, that non-termination is not considered to be an error. We define $\texttt{ok}[\![\texttt{A}]\!] : \mathbb{D} \to \mathbb{D}$ and $\texttt{err}[\![\texttt{A}]\!] : \mathbb{D} \to \mathbb{D}$ to collect respectively regular and error states in the evaluation of `A`.

$$\texttt{ok}[\![\texttt{A}]\!]R \triangleq R \circ \{\, ((m, i, r), (m[\texttt{x} \leftarrow \mathcal{A}[\![\texttt{A}]\!]m], i, r)) \mid (m, i, r) \in \mathbb{S}, \mathcal{A}[\![\texttt{A}]\!]m \neq \lightning \,\} \tag{7.8}$$

$$\texttt{err}[\![\texttt{A}]\!]R \triangleq R \circ \{\, ((m, i, r), (m[\texttt{ret} \leftarrow 1], i, r)) \mid (m, i, r) \in \mathbb{S}, \mathcal{A}[\![\texttt{A}]\!]m = \lightning \,\} \tag{7.9}$$

$$\mathcal{S}[\![\texttt{x = A}]\!](R, E) \triangleq (\texttt{ok}[\![\texttt{A}]\!]R, E \cup \texttt{err}[\![\texttt{A}]\!]R) \tag{7.10}$$

We define $\texttt{test}[\![\texttt{B}]\!] : \mathbb{D} \to \mathbb{D}$ to filter states according to a boolean condition `B`:

$$\texttt{test}[\![\texttt{B}]\!]R \triangleq R \circ \{((m, i, r), (m, i, r)) \mid (m, i, r) \in \mathbb{S}, \mathcal{B}[\![\texttt{B}]\!]m = \texttt{tt}\} \tag{7.11}$$

We abuse the notation, and we use $\texttt{err}[\![\texttt{B}]\!]$ for the boolean evaluation of errors.

$$\begin{aligned} \mathcal{S}[\![\texttt{if (B) } \texttt{S}_t \texttt{ else } \texttt{S}_e]\!](R, E) \triangleq\ & \textbf{let } (R_t, E_t) = \mathcal{S}[\![\texttt{S}_t]\!](\texttt{test}[\![\texttt{B}]\!]R, E) \textbf{ in} \\ & \textbf{let } (R_e, E_e) = \mathcal{S}[\![\texttt{S}_e]\!](\texttt{test}[\![\neg\texttt{B}]\!]R, E) \textbf{ in} \\ & (R_t \cup R_e, E_t \cup E_e \cup \texttt{err}[\![\texttt{B}]\!]R) \end{aligned} \tag{7.12}$$

The semantics of while statements is a classic fixpoint definition. The operator $\dot{\cup}$ denotes the point-wise set union on pairs. We denote the point-wise lifting of an operator $\diamond$ to tuples as $\dot{\diamond}$.

$$\begin{aligned} \mathcal{S}[\![\texttt{while (B) } \mathtt{S}_b]\!](R, E) \triangleq \textbf{let } (R_f, E_f) = \mathrm{lfp}\, F \textbf{ in } (\texttt{test}[\![\neg \mathtt{B}]\!] R_f, E_f) \\ \textbf{where } F(R_1, E_1) \triangleq (R, E) \dot{\cup} \mathcal{S}[\![\texttt{if (B) } \mathtt{S}_b \texttt{ else skip}]\!](R_1, E_1) \end{aligned} \tag{7.13}$$

Example 7.1 (Nondeterminism and random input read)
Consider the following program:

```
1 x = rand();
2 if (x == 0) {
3   y = 1;
4 } else {
5   y = input();
6 }
7 z = 1 / y;
```

The program sets the variable y to 1 or to a dynamic user input read depending on a nondeterministic choice. Then, 1/y is assigned to the variable z. If the user input is read and the user inserts 0, then the program results in a runtime error. The semantics $\mathcal{S}[\![\mathtt{P}]\!]$ of the program can be partitioned into three groups of pre-post states. First, there are the input-output pairs where the first number in the random sequence is zero. Here, the final values of y and z are 1. Second, there is the set of input-output pairs where the first numbers in both the random and the input queues are non-zero. In this group, the output values for y and z are respectively the first number in the input sequence, and 1 divided by that number. Lastly, there are the input-output pairs where the first number in the random sequence is not zero, but the first number in the input sequence is zero. Here y is 0 in the output state, but z is not changed from its initial value. On the other hand, unlike the other cases, the variable ret is set to 1.

7.6. Safety-nonexploitability

In this section, we first give a formal definition of *safety-nonexploitability* as a 2-hypersafety property [50] (see Section 6.3.2). Then, we put forward an alternative characterization based on semantically tainted (i.e. user-controlled) variables, which we leverage to introduce a sound, effective analysis for safety-nonexploitability. The proofs of the theoretical results are reported in Appendix A.

Safety-nonexploitability formalizes the idea that by modifying only the user input at the beginning of a program, it is not possible to change whether the program results in a runtime error or not. Since we designed our concrete semantics to explicitly represent runtime errors as states with the return variable set to 1, we use program memories to differentiate erroneous states from regular ones.

Definition 7.1 (Safety-nonexploitability)

$$\mathcal{NE} \in \wp(\mathbb{D})$$
$$\mathcal{NE} \triangleq \{\, R \in \mathbb{D} \mid \forall((m_0, i_0, r_0), (m_1, i_1, r_1)), ((m'_0, i'_0, r'_0), (m'_1, i'_1, r'_1)) \in R : m_0 = m'_0, r_0 = r'_0 \implies m_1(\texttt{ret}) = m'_1(\texttt{ret}) \,\}$$

Observe that our definition of safety-nonexploitability does not explicitly require user inputs at the beginning of the program to be different. Nevertheless, since our semantics is deterministic, if two states are exactly equal at the beginning of the program, they result in the same state at the end, and therefore the condition $m_1(\texttt{ret}) = m'_1(\texttt{ret})$ trivially holds.

Remark 7.1 ($\mathcal{NE}$ is a 2-hypersafety property)
According to the taxonomy of program properties that we presented in Section 6.3, safety-nonexploitability is a 2-hypersafety property (see Def. 6.4). This is due to the fact that to disprove $\mathcal{NE}$ it is sufficient to find two input-output pairs that in the initial state agree on everything but the input sequence, and differ in the output value for `ret`.

Safety-exploitability is defined as the negation of safety-nonexploitability. We formally define the property because we often mention it in our examples, even

though in this work we are interested in proving safety-nonexploitability.

Definition 7.2 (Safety-exploitability)

$$\mathscr{E} \in \wp(\mathbb{D})$$
$$\mathscr{E} \triangleq \{ R \in \mathbb{D} \mid \exists((m_0, i_0, r_0), (m_1, i_1, r_1)), ((m'_0, i'_0, r'_0), (m'_1, i'_1, r'_1)) \in R : m_0 = m'_0, r_0 = r'_0 : m_1(\texttt{ret}) \neq m'_1(\texttt{ret}) \}$$

Example 7.2 (Nonexploitability and exploitability)
According to our definition, the following program is safety-exploitable:

```
x = input(); 1/x
```

This is because if we consider two initial states, one in which the first element of the input sequence is zero, and the other in which it is not, we observe that the value of `ret` changes. Conversely, the following program is safety-nonexploitable:

```
x = rand(); 1/x
```

Even if there is a possible division by zero, once we fix the sequence of random numbers, changing the user input does not result in modifying the value of `ret`. If we did not compare pairs of initial states with the *same* sequence of random numbers, the program would be exploitable, even if the user input is never read.

Example 7.3 (Comparison with robust reachability [213])
A bug is *robustly reachable* if there exists a user input for which the bug is always reached, regardless of the value of the random input. Consider a program that always results in a division by zero: `1/0`. The program is safety-nonexploitable: for any possible user input, the value of `ret` will always be 1. Conversely, the error is robustly reachable, as it is trivially reached for *any* user input. This highlights an important difference between the two concepts: safety-nonexploitability requires the user input to be *effectively used* in triggering program errors, while robust reachability does not.

In what follows, we show that safety-nonexploitability can be expressed in terms of *semantically tainted variables*. Intuitively, a variable is tainted if an attacker can control its value. Taint analysis [202], informally introduced in Section 7.3, is a well-known technique in computer security to track the variables that are controlled by external users. However, many existing approaches use heuristics and syntactic formulations of the problem, which may be both imprecise and unsound. In contrast, we rely on a *semantic* approach, which is grounded in the formal semantics of programs. The following hyperproperty captures the set of semantics where the value of a variable x depends on the user's input, i.e. x is tainted. The definition formalizes the intuition that x is tainted if, by modifying only the user input, it is possible to change the value of x.

Definition 7.3 (Taint)
Let $\mathrm{x} \in \mathbb{V}$.

$$\mathscr{T} : \mathbb{V} \to \wp(\mathbb{D})$$
$$\mathscr{T}(\mathrm{x}) \triangleq \{R \in \mathbb{D} \mid \exists((m_0, i_0, r_0), (m_1, i_1, r_1)), ((m_0', i_0', r_0'), (m_1', i_1', r_1')) \in R : m_0 = m_0', r_0 = r_0' : m_1(\mathrm{x}) \neq m_1'(\mathrm{x})\}$$

We define abstraction and concretization functions for $\wp(\mathbb{V})$.

$$\alpha_t : \wp(\mathbb{D}) \to \wp(\mathbb{V}) \qquad \gamma_t : \wp(\mathbb{V}) \to \wp(\mathbb{D})$$
$$\alpha_t(\mathscr{R}) \triangleq \{\mathrm{x} \in \mathbb{V} \mid \mathscr{R} \subseteq \mathscr{T}(\mathrm{x})\} \qquad \gamma_t(T) \triangleq \bigcap_{\mathrm{x} \in T} \mathscr{T}(\mathrm{x})$$

There is a Galois connection between $\wp(\mathbb{D})$ and $\wp(\mathbb{V})$ defined by α_t and γ_t:

$$(\wp(\mathbb{D}), \subseteq) \underset{\alpha_t}{\overset{\gamma_t}{\leftrightarrows}} (\wp(\mathbb{V}), \supseteq) \tag{7.14}$$

The order for the abstract domain $\wp(\mathbb{V})$ is $\supseteq$ because if we consider more relations, we obtain fewer tainted variables common to *all* of these relations. Notice that this is different from observing that larger relations present more tainted variables, which will be discussed later in this section. By relying on $\mathscr{T}$, we can formally define when a variable is tainted in a program.

Definition 7.4 (Semantically tainted variable)
A variable `x` is *tainted* in a program `P` if $\texttt{x} \in \alpha_t(\{\mathcal{S}[\![\texttt{P}]\!]\})$.

Example 7.4 (Implicit flows and taint)
As already mentioned in Section 7.3, if statements can generate *implicit flows* [126], namely dependencies that arise from the program control flow. Consider the following program:

```
x = input(); if (x==0) y = 1 else y = 2
```

Depending on the user's input, `y` can be either 1 or 2, and accordingly to our semantic characterization of tainted variables, `y` is tainted. Taint analyzers (e.g. [214, 201, 127, 194]) often ignore implicit flows, considering only *explicit flows* (i.e., when tainting is propagated through assignments only), which is unsound in our framework. In the analysis described in Section 7.8, we develop an abstraction that does take implicit flows into account. Observe that in our formal definition of taint–which is based on the semantics of a program–there is no need to differentiate implicit and explicit flows.

If the user cannot control the value of `ret`, then she cannot control whether there is a runtime error, i.e. the program is safety-nonexploitable. This is the fundamental observation used in the following alternative characterization of $\mathscr{NE}$.

Theorem 7.1 (Chaterization of $\mathscr{NE}$ with taint)
Let $R \in \mathbb{D}$.

$$R \in \mathscr{NE} \iff \texttt{ret} \notin \alpha_t(\{R\})$$

Thm. 7.1 is significant because it shows that safety-nonexploitability can be verified by relying on a taint analysis. In contrast to classic taint analyses, simply tracking the set of user-controlled variables is not sufficient, as to infer whether `ret` is tainted we also need to detect runtime errors. In fact, `ret` does not syntactically appear in programs, and its value changes only when program failures occur. To determine when this happens, is it important to consider the *values* of the variables.

Without semantic information about the values of the variables, every expression with a division should be considered dangerous in order to be sound, and this would result in an unacceptable loss of precision. In Section 7.8 we put forward a sound analysis by abstract interpretation that can prove programs to be safety-nonexploitable by combining a classic value analysis with a taint analysis. The former detects program locations that potentially present runtime errors, while the latter determines whether the user can trigger those errors.

Hyperproperties verification is challenging for analyses based on abstract interpretation, because not every hyperproperty is *subset-closed* [50]: by computing an overapproximation R_1 of R_0, the fact that R_1 respects an hyperproperty does not, in the general case, imply that R_0 respects the hyperproperty. To overcome this problem, many works rely on *hypersemantics* [215, 216, 217, 218, 219]: the concrete semantics of a program is a set of sets of states, in contrast to a classic set of states. The main disadvantage of hypersemantics is that *hyperdomains* [215, 216, 217] are incompatible with regular abstract domains: the former abstract hypersemantics, while the latter abstract regular semantics.

In this work, we rely on the standard abstract interpretation framework. In the rest of this section, we show that an overapproximation of the concrete semantics is sufficient to prove safety-nonexploitability. In particular, we show that $\mathscr{NE}$ is subset-closed, and the consequence is that an overapproximation of $\mathbb{S}[\![\mathtt{S}]\!]$ is enough to prove safety-nonexploitabilty. A significant benefit of using the standard framework is that we can combine a taint analysis with any existing over-approximating value domain, which leads to a modular design. Furthermore, enhancing the precision of the numeric analysis improves the precision of the taint analysis as well. Observe that, as discussed in this section, in our context, it is important to rely on a classic safety analysis (and hence, on regular abstract numeric domains) to identify expressions that potentially present runtime errors.

We observe that larger semantics have more tainted variables. This holds due to the existential quantifier in Def. 7.3. Let $R_0, R_1 \in \mathbb{D}$.

$$R_0 \subseteq R_1 \implies \alpha_t(\{R_0\}) \subseteq \alpha_t(\{R_1\}) \tag{7.15}$$

By using this result, we observe that if `ret` is not tainted in R_1, it cannot be tainted in R_0. This implies that if R_1 is safety-nonexploitable, then R_0 is safety-

nonexploitable, namely $\mathscr{NE}$ is subset-closed.

Theorem 7.2 ($\mathscr{NE}$ is subset-closed)
Let $R_0, R_1 \in \mathbb{D}$.

$$(R_0 \subseteq R_1 \text{ and } R_1 \in \mathscr{NE}) \implies R_0 \in \mathscr{NE}$$

Thm. 7.2 is significant because it implies that by overapproximating the semantics of a program, we can still prove that it is safety-nonexploitable. This justifies why the standard abstract interpretation framework is sufficient, and allows using the large library of existing abstract value domains. The theorem formalizes the intuition that if it is not possible for an attacker to trigger any runtime error, by further reducing the semantics of the program–and hence the capabilities of the attacker and the set of errors–she is still not able to make the program fail.

7.7. Taint concrete semantics

In this section, we define the non-computable concrete taint semantics that we overapproximate in Section 7.8. The semantics associates the reachable states with the set of semantically tainted variables using the abstraction function α_t. As the semantics is not *structural* (i.e. defined by induction on the program syntax), we also develop a structural equivalent definition. This is necessary in order to overapproximate the concrete taint semantics with an inductive and effectively computable abstract semantics.

We first define the reachability taint semantics of statements $\mathcal{S}_t[\![\mathtt{S}]\!] : (\mathbb{D} \times \mathbb{D}) \to (\mathbb{D} \times \mathbb{D} \times \wp(\mathbb{V}))$. This semantics associates each statement with its set of truly tainted variables by relying on α_t.

$$\mathcal{S}_t[\![\mathtt{S}]\!](R, E) \triangleq \textbf{let } (R_1, E_1) = \mathcal{S}[\![\mathtt{S}]\!](R, E) \textbf{ in } (R_1, E_1, \alpha_t(\{\, R_1 \,\})) \tag{7.16}$$

Example 7.5 (Taint concrete semantics)
Consider the following program:

```
x = rand(); if (x == 0) {y = input()}
```

```c
void main() {
  if (getchar() == 'a')
    rand();
  int z = rand();
}
```

FIGURE 7.3. C program that reads pseudo-random numbers

If we consider its concrete taint semantics starting from the initial states I, we find that the variable y is tainted in $\mathcal{S}_t[\![\mathtt{S}]\!](I, \emptyset)$. This is due to the fact that there exist two pairs of initial states that agree on everything but the input sequence and result in different values for y. For instance, the initial states $(\{\mathtt{x} \mapsto 0, \mathtt{y} \mapsto 0\}, 0^\infty, 0^\infty)$ and $(\{\mathtt{x} \mapsto 0, \mathtt{y} \mapsto 0\}, 1^\infty, 0^\infty)$ result respectively in 1 and 0 for y.

We then define the reachability taint semantics for programs $\mathcal{S}_t[\![\mathtt{P} := \mathtt{S}]\!] \in \mathbb{D} \times \wp(\mathbb{V})$. As regular and erroneous states are merged at the end of programs, we use α_t to obtain the tainted variables in the union.

$$\mathcal{S}_t[\![\mathtt{P} := \mathtt{S}]\!] \triangleq \textbf{let } (R, E) = \mathcal{S}[\![\mathtt{S}]\!](I, \emptyset) \textbf{ in } (R \cup E, \alpha_t(\{R \cup E\})) \tag{7.17}$$

Observe that only at the end of the program `ret` can become tainted: regular and erroneous states are partitioned in the semantics for statements, so that `ret` is always constant (0 for the normal executions and 1 for the others). The program P is then safety-nonexploitable iff `ret` is not tainted in $\mathcal{S}_t[\![\mathtt{P} := \mathtt{S}]\!]$.

Remark 7.2 (Taint and `rand` statements)
The statement `x = rand()` can taint x if there are two executions in which the sequence of random numbers is out-of-sync due to a user action. This is because in the definition of $\mathcal{T}$ we compare pairs of execution with the *same* sequence of random numbers. Consider the following program:

```c
x = input();
if (x != 0) {
  y = rand();
}
z = rand();
```

The user can control whether z is assigned to the first or the second number in the random sequence. If we fix as random sequence 1, 2, . . . , we can observe that z can be either 1 or 2 at the end of the program, depending on the user's input.
Observe that this behaviour is relevant in scenarios where the attacker has partial knowledge about the uncontrolled random input. For instance, consider the program in Figure 7.3, where the application first reads a character from standard input. If the character is a, the program reads the first pseudo-random number, and then it assigns z to `rand()`. As the sequence of random numbers has not been initialized, it does not change and could be predicted across different executions. The user can make the program assign z to the first or second number in the sequence, being able to influence the assigned value. Another relevant case is when a program reads a file with unmodifiable but public content. If an attacker can control which bytes are read, then she can influence the execution of the program without even modifying the contents of the file.
The fact that random read statements can potentially taint the assigned variable directly derives from our semantic definition of $\mathscr{T}$. By changing the definition of $\mathscr{T}$, it is possible to choose whether random read statements can taint assigned variables. We make the choice to use random read statements as potential sources of tainted data because, in a context in which an attacker has (partial) knowledge about the unmodifiable pseudo-random input, such statements can be exploited to influence the execution of the program. While it would be possible to support the classic model where the attacker has no knowledge about the random input, this is less interesting in a context where security is considered fundamental.

As we want to overapproximate the concrete taint semantics by induction on the program structure, we give a structural equivalent definition of $\mathcal{S}_t[\![\mathtt{S}]\!]$. The non-computable semantics $\hat{\mathcal{S}}_t[\![\mathtt{S}]\!] : (\mathbb{D} \times \mathbb{D} \times \wp(\mathbb{V})) \to (\mathbb{D} \times \mathbb{D} \times \wp(\mathbb{V}))$ inductively collects the truly tainted variables. The semantics takes as additional input parameter the set of previously tainted variables, which are used to infer the set of tainted variables after the execution of the statement. As usual, `skip` statements do not

modify the input.

$$\hat{\mathcal{S}}_t[\![\texttt{skip}]\!](R, E, T) \triangleq (R, E, T) \tag{7.18}$$

After the execution of `x = input()`, `x` is tainted if and only if the user can provide two different numbers as the first element of the sequence.

$$\begin{aligned}
&\hat{\mathcal{S}}_t[\![\texttt{x = input()}]\!](R, E, T) \triangleq \qquad (7.19)\\
&\quad \textbf{let } (R_1, E_1) = \mathcal{S}[\![\texttt{x = input()}]\!](R, E) \textbf{ in}\\
&\quad \textbf{let } T_1 = \{\, \texttt{y} \in T \mid \texttt{y} \neq \texttt{x} \,\} \cup\\
&\qquad \{\, \texttt{x} \mid \exists((m_0, i_0, r_0), (m_1, i_1, r_1)), ((m'_0, i'_0, r'_0), (m'_1, i'_1, r'_1)) \in R :\\
&\qquad\quad m_0 = m'_0, r_0 = r'_0 : \texttt{hd}(i_1) \neq \texttt{hd}(i'_1) \,\} \textbf{ in}\\
&\quad (R_1, E_1, T_1)
\end{aligned}$$

Random read statements follow the same pattern.

$$\begin{aligned}
&\hat{\mathcal{S}}_t[\![\texttt{x = rand()}]\!](R, E, T) \triangleq \qquad (7.20)\\
&\quad \textbf{let } (R_1, E_1) = \mathcal{S}[\![\texttt{x = rand()}]\!](R, E) \textbf{ in}\\
&\quad \textbf{let } T_1 = \{\, \texttt{y} \in T \mid \texttt{y} \neq \texttt{x} \,\} \cup\\
&\qquad \{\, \texttt{x} \mid \exists((m_0, i_0, r_0), (m_1, i_1, r_1)), ((m'_0, i'_0, r'_0), (m'_1, i'_1, r'_1)) \in R :\\
&\qquad\quad m_0 = m'_0, r_0 = r'_0 : \texttt{hd}(r_1) \neq \texttt{hd}(r'_1) \,\} \textbf{ in}\\
&\quad (R_1, E_1, T_1)
\end{aligned}$$

Observe that the scenario where `x` becomes tainted after the execution of `x = rand()` can only occur if there exist two executions that only differ in the input sequence, resulting in one of the two reading from the random sequence at least one additional time, as the initial random sequences are equal. This happens when the user can control how many times there is a read from the sequence of random numbers (see Remark 7.2).

For assignments, we taint the assigned variable if the outcome of the arithmetic evaluation can be controlled by the user. This happens when the tainted variables in the arithmetic expressions can change the outcome of the evaluation. Observe that runtime errors are handled by the underlying $\mathcal{S}[\![\mathtt{S}]\!]$ semantics. Since they do

not influence the set of tainted variables after the execution of the assignment, runtime errors are ignored to compute the set of truly tainted variables.

$$
\begin{aligned}
&\hat{\mathcal{S}}_t[\![\mathtt{x = A}]\!](R, E, T) \triangleq \\
&\quad \textbf{let } (R_1, E_1) = \mathcal{S}[\![\mathtt{x = A}]\!](R, E) \textbf{ in} \\
&\quad \textbf{let } T_1 = \{\, \mathtt{y} \in T \mid \mathtt{y} \neq \mathtt{x} \,\} \cup \\
&\qquad \{\, \mathtt{x} \mid \exists((m_0, i_0, r_0), (m_1, i_1, r_1)), ((m'_0, i'_0, r'_0), (m'_1, i'_1, r'_1)) \in R : \\
&\qquad\quad m_0 = m'_0, r_0 = r'_0 : \lightning \neq \mathcal{A}[\![\mathtt{A}]\!]m_1 \neq \mathcal{A}[\![\mathtt{A}]\!]m'_1 \neq \lightning \,\} \textbf{ in} \\
&\quad (R_1, E_1, T_1)
\end{aligned}
\tag{7.21}
$$

As usual, the semantics of statement composition $\mathtt{S}_1\,;\,\mathtt{S}_2$ is defined as the execution of $\mathtt{S}_2$ on the result of the execution of $\mathtt{S}_1$.

$$
\hat{\mathcal{S}}_t[\![\mathtt{S}_1\,;\,\mathtt{S}_2]\!](R, E, T) \triangleq \hat{\mathcal{S}}_t[\![\mathtt{S}_2]\!](\hat{\mathcal{S}}_t[\![\mathtt{S}_1]\!](R, E, T)) \tag{7.22}
$$

As discussed in this chapter, if statements can generate *implicit flows* [126] (see Example 7.4). We define a helper function to compute the set of variables that are tainted due to implicit flows. This function considers pairs of executions that initially differ only by the user input. If two executions follow different branches and yield different values for a variable, such a variable is tainted. This happens when the evaluation of the boolean condition depends on the user input.

$$
\begin{aligned}
&\mathtt{diff}[\![\mathtt{if\ (B)\ S}_t\ \mathtt{else\ S}_e]\!] : \mathbb{D} \to \wp(\mathbb{V}) \\
&\mathtt{diff}[\![\mathtt{if\ (B)\ S}_t\ \mathtt{else\ S}_e]\!]R \triangleq \\
&\quad \{\, \mathtt{x} \in \mathbb{V} \mid \exists((m_0, i_0, r_0), (m_1, i_1, r_1)), ((m'_0, i'_0, r'_0), (m'_1, i'_1, r'_1)) \in R : \\
&\qquad \textbf{let } (R_t, E_t) = \mathcal{S}[\![\mathtt{S}_t]\!](\mathtt{test}[\![\mathtt{B}]\!]\{((m_0, i_0, r_0), (m_1, i_1, r_1))\}, \emptyset) \textbf{ in} \\
&\qquad \textbf{let } (R_e, E_e) = \mathcal{S}[\![\mathtt{S}_e]\!](\mathtt{test}[\![\neg\mathtt{B}]\!]\{((m'_0, i'_0, r'_0), (m'_1, i'_1, r'_1))\}, \emptyset) \textbf{ in} \\
&\qquad \exists(m_2, i_2, r_2) : ((m_0, i_0, r_0), (m_2, i_2, r_2)) \in R_t : \\
&\qquad \exists(m'_2, i'_2, r'_2) : ((m'_0, i'_0, r'_0), (m'_2, i'_2, r'_2)) \in R_e : \\
&\qquad m_0 = m'_0, r_0 = r'_0 : \mathcal{B}[\![\mathtt{B}]\!]m_1 = \mathtt{tt}, \mathcal{B}[\![\mathtt{B}]\!]m'_1 = \mathtt{ff} : \\
&\qquad m_2(\mathtt{x}) \neq m'_2(\mathtt{x}) \,\}
\end{aligned}
\tag{7.23}
$$

Example 7.6 (Semantically tainted variables and implicit flows)
Consider the following program:

```
if (x == 0) { y = x } else { y = 0 }
```

Consider the case in which x is tainted. While it might seem like there is an implicit flow from x to y, this is not actually the case. In fact, even if y is assigned in a branch whose execution depends on a tainted variable, the value of y will always be 0 at the end of the program. Sound taint analyses often have to classify y as tainted, as they do not usually compare the output values of the variables after the execution of the branches. Our definition of $\texttt{diff}[\![\texttt{if (B) } \mathtt{S}_t \texttt{ else } \mathtt{S}_e]\!]$ is strictly semantic, and does not classify y as tainted.

The tainted variables in if statements are obtained as the union of the tainted variables in the individual branches, to which we add the tainted variables due to implicit flows computed by $\texttt{diff}[\![\texttt{if (B) } \mathtt{S}_t \texttt{ else } \mathtt{S}_e]\!]$.

$$
\begin{aligned}
&\hat{\mathbb{S}}_t[\![\texttt{if (B) } \mathtt{S}_t \texttt{ else } \mathtt{S}_e]\!](R, E, T) \triangleq \\
&\quad \textbf{let } (R_t, E_t, T_t) = \hat{\mathbb{S}}_t[\![\mathtt{S}_t]\!](\texttt{test}[\![\mathtt{B}]\!]R, E, T \setminus \overline{\alpha_t(\{\texttt{test}[\![\mathtt{B}]\!]R\})}) \textbf{ in} \\
&\quad \textbf{let } (R_e, E_e, T_e) = \hat{\mathbb{S}}_t[\![\mathtt{S}_e]\!](\texttt{test}[\![\neg\mathtt{B}]\!]R, E, T \setminus \overline{\alpha_t(\{\texttt{test}[\![\neg\mathtt{B}]\!]R\})}) \textbf{ in} \\
&\quad \textbf{let } T_{te} = \texttt{diff}[\![\texttt{if (B) } \mathtt{S}_t \texttt{ else } \mathtt{S}_e]\!]R \textbf{ in} \\
&\quad (R_t \cup R_e, E_t \cup E_e \cup \texttt{err}[\![\mathtt{B}]\!]R, T_t \cup T_e \cup T_{te})
\end{aligned}
\tag{7.24}
$$

Example 7.7 (Semantically tainted variables and boolean conditions)
Observe that the tainted variables inside of the individual branches correspond to the previously tainted variables to which we remove the not tainted ones (namely $\overline{\alpha_t(\{\texttt{test}[\![\mathtt{B}]\!]R\})}$) after the execution of the boolean condition. Consider the following program:

```
1 x = input()
2 if (x == 0) {
3   ...
4 }
```

Even if x is tainted before executing the condition at line 2, inside of the then

branch at line 3 the variable is not tainted. The reason is that at line 3, x is 0, namely a constant whose value cannot be controlled by the user. Our rule captures the fact that x is not tainted inside of the then branch. Let R be the set of reachable states before line 2, and consider $\mathbb{V} = \{\mathtt{x}\}$:

$$R = \{\, ((m_0, i_0, r_0), (m_1, i_1, r_1)) \mid m_1(\mathtt{x}) = \mathtt{hd}(i_0), i_1 = \mathtt{tl}(i_0) \,\}$$

Then, the set of tainted variables at line 3 corresponds to the following:

$$\begin{aligned}
&\{\mathtt{x}\} \setminus \overline{\alpha_t(\{\mathtt{test}[\![\mathtt{x == 0}]\!]R\})} \\
&= \{\mathtt{x}\} \setminus \overline{\alpha_t(\{\, ((m_0, i_0, r_0), (m_1, i_1, r_1)) \mid m_1(\mathtt{x}) = 0, \mathtt{hd}(i_0) = 0, i_1 = \mathtt{tl}(i_0) \,\})} \\
&= \{\mathtt{x}\} \setminus \overline{\emptyset} \\
&= \{\mathtt{x}\} \setminus \{\mathtt{x}\} \\
&= \emptyset
\end{aligned}$$

Observe that even if x is not tainted inside the then branch, any variable that is semantically assigned within it will be tainted at the end of the if statement due to an implicit flow. Such a dependency is captured by $\mathtt{diff}[\![\mathtt{if\ (B)\ S}_t\ \mathtt{else\ S}_e]\!]$.

The taint semantics for while statements is expressed as a least fixpoint.

$$\begin{aligned}
\hat{\mathbb{S}}_t[\![\mathtt{while\ (B)\ S}_b]\!](R, E, T) \triangleq\ & \textbf{let } (R_f, E_f, T_f) = \mathrm{lfp}\, F \textbf{ in} \\
& (\mathtt{test}[\![\neg \mathtt{B}]\!]R_f, E_f, T_f \setminus \overline{\alpha_t(\{\mathtt{test}[\![\neg \mathtt{B}]\!]R_f\})}) \\
\textbf{where } F(R_1, E_1, T_1) \triangleq\ & (R, E, T) \dot{\cup} \hat{\mathbb{S}}_t[\![\mathtt{if\ (B)\ S}_b\ \mathtt{else\ skip}]\!](R_1, E_1, T_1)
\end{aligned} \tag{7.25}$$

The following theorem formalizes that the structural and non-structural taint semantics are equivalent. The theorem holds under the assumption that the input parameter T of the structural version exactly corresponds to the set of tainted variables in the reachable states.

Theorem 7.3 (Correctness of $\hat{\mathbb{S}}_t[\![\mathtt{S}]\!]$)

Let $R, E \in \mathbb{D}$ and $T \in \wp(\mathbb{V})$ such that $T = \alpha_t(\{R\})$.

$$\mathcal{S}_t[\![\mathtt{S}]\!](R, E) \doteq \hat{\mathcal{S}}_t[\![\mathtt{S}]\!](R, E, T)$$

Thm. 7.3 is significant as it shows that the structural and the non-structural semantics are equivalent under the assumption that the input of the structural version exactly corresponds to the set of tainted variables in the reachable states. At the beginning of the program, this set can be initialized to $\emptyset$, as there are no tainted variables in the initial states.

7.8. Taint abstract semantics

In this section, we introduce a computable sound overapproximation of the concrete taint semantics presented in Section 7.7. This abstraction of the concrete non-computable semantics is parametric in the underlying abstract domain used to overapproxiamate the values of the variables. In contrast to traditional techniques, we leverage numeric invariants to improve the precision of the taint analysis.

Let $\mathbb{D}^\sharp$ be the abstract numeric domain used to overapproximate $\mathbb{D}$, and $\gamma_d : \mathbb{D}^\sharp \to \mathbb{D}$ be the concretization function. The domain $\mathbb{D}^\sharp$ is equipped with partial order $\subseteq_d^\sharp$ and abstract join $\cup_d^\sharp$, while $\bot_d^\sharp$ is the bottom element. We assume $\mathcal{S}_d^\sharp[\![\mathtt{S}]\!] : (\mathbb{D}^\sharp \times \mathbb{D}^\sharp) \to (\mathbb{D}^\sharp \times \mathbb{D}^\sharp)$ given by the numeric domain to be a sound computable abstraction of $\mathcal{S}[\![\mathtt{S}]\!]$:

$$\forall R^\sharp, E^\sharp \in \mathbb{D}^\sharp : \mathcal{S}[\![\mathtt{S}]\!](\dot{\gamma}_d(R^\sharp, E^\sharp)) \dot{\subseteq} \dot{\gamma}_d(\mathcal{S}_d^\sharp[\![\mathtt{S}]\!](R^\sharp, E^\sharp)) \tag{7.26}$$

Where $\dot{\gamma}_d$ is the point-wise application of the concretization function γ_d to pairs of abstract elements. The abstract value domain also exposes the abstract functions $\mathtt{test}^\sharp[\![\mathtt{B}]\!]$ and $\mathtt{err}^\sharp[\![\mathtt{B}]\!]$ to overapproximate the concrete ones.

Remark 7.3 (Numeric domains and taint analysis)
While the concrete semantics is defined as a set of input-output relations to express safety-nonexploitability, in the numeric abstraction it is possible to use numeric domains that simply abstract sets of states. In order to do this, it is sufficient to abstract only the image of the relations, and then consider

each possible state as initial in the concretization.
Our abstract taint semantics achieves enhanced precision by querying the numeric domain. Nevertheless, the queries are limited to properties of the reachable states (e.g., constancy of a variable), so that the numeric information about the initial states does not improve the precision. This implies that by using regular numeric domains (such as those presented in Section 6.4.3) that do not keep information about initial states, we do not lose precision.

In the rest of the section, we structurally define the *abstract taint semantics* $\mathbb{S}_t^\sharp[\![\mathtt{S}]\!] : (\mathbb{D}^\sharp \times \mathbb{D}^\sharp \times \wp(\mathbb{V})) \to (\mathbb{D}^\sharp \times \mathbb{D}^\sharp \times \wp(\mathbb{V}))$. The semantics collects an overapproximation of the reachable states, the error states, and the tainted variables. The concretization function $\gamma : (\mathbb{D}^\sharp \times \mathbb{D}^\sharp \times \wp(\mathbb{V})) \to (\mathbb{D} \times \mathbb{D} \times \wp(\mathbb{V}))$ is defined as follows:

$$\gamma(R^\sharp, E^\sharp, T^\sharp) \triangleq (\gamma_d(R^\sharp), \gamma_d(E^\sharp), T^\sharp) \tag{7.27}$$

The soundness theorem states that the abstract semantics exhibits more tainted variables than those in the concrete semantics $\hat{\mathbb{S}}_t[\![\mathtt{S}]\!]$.

Theorem 7.4 (Soundness of $\mathbb{S}_t^\sharp[\![\mathtt{S}]\!]$)
Let $R^\sharp, E^\sharp \in \mathbb{D}^\sharp$ and $T^\sharp \in \wp(\mathbb{V})$.

$$\hat{\mathbb{S}}_t[\![\mathtt{S}]\!](\gamma(R^\sharp, E^\sharp, T^\sharp)) \dot{\subseteq} \gamma(\mathbb{S}_t^\sharp[\![\mathtt{S}]\!](R^\sharp, E^\sharp, T^\sharp))$$

In our abstract semantics, we taint `ret` every time there is a *possible* runtime error due to user input. This ensures that if `ret` is not tainted in $\mathbb{S}_t^\sharp[\![\mathtt{S}]\!]$, it will not be tainted at the end of the program, i.e. the program is safety-nonexploitable. Let $I^\sharp \in \mathbb{D}^\sharp$ be an overapproximation of the set of initial states, namely $I \subseteq \gamma_d(I^\sharp)$.

Theorem 7.5 (Soundness of the safety-nonexploitability analysis)
Let $\mathtt{P} := \mathtt{S}$ be a program, and let $(R^\sharp, E^\sharp, T^\sharp) = \mathbb{S}_t^\sharp[\![\mathtt{S}]\!](I^\sharp, \bot_d^\sharp, \emptyset)$.

$$\mathtt{ret} \notin T^\sharp \implies \mathbb{S}[\![\mathtt{P} := \mathtt{S}]\!] \in \mathcal{NE}$$

In the rest of this section, we define by structural induction $\mathbb{S}_t^\sharp[\![\mathtt{S}]\!]$. The abstract semantics collects an overapproximation of the tainted variables, and specifically

taints `ret` whenever a runtime error potentially caused by the user occurs. We will take advantage of the helper function $\texttt{taint}^\sharp[\![\texttt{A}]\!] : (\mathbb{D}^\sharp \times \wp(\mathbb{V})) \to \mathbb{B}$ that returns `tt` if the result of the evaluation of `A` could depend on tainted variables. The classic well-known approach to implement $\texttt{taint}^\sharp[\![\texttt{A}]\!]$ is to return `tt` if a tainted variable syntactically occurs in `A`. This is sound because if there is no user-controlled variable in `A`, then the user cannot influence the outcome of the evaluation. Nevertheless, the approach can sometimes result in a loss of precision. For instance, consider the program `x = input(); y = x; z = x-y`. The user cannot control the value of `z`, as it is always 0. By using a relational abstract domain such as polyhedra or octagons [158], it is possible to determine that `z` is constant, and therefore that it is not tainted. We rely on the function $\texttt{isconst}^\sharp[\![\texttt{A}]\!] : \mathbb{D}^\sharp \to \mathbb{B}$ provided by the numeric domain that returns `tt` if the evaluation of `A` in an abstract state is constant. The implementation depends on the numeric domain, and for intervals can be implemented as follows:

$$\texttt{isconst}_i^\sharp[\![\texttt{A}]\!]R^\sharp \triangleq \begin{cases} \texttt{tt} & \text{if } \mathcal{A}^\sharp_{\mathbb{V}^\sharp_{\mathbb{I}}}[\![\texttt{A}]\!]R^\sharp = \bot_i \\ l = u & \text{if } \mathcal{A}^\sharp_{\mathbb{V}^\sharp_{\mathbb{I}}}[\![\texttt{A}]\!]R^\sharp = [l, u] \end{cases} \tag{7.28}$$

We can then define $\texttt{taint}^\sharp[\![\texttt{A}]\!]$ as follows:

$$\texttt{taint}^\sharp[\![\texttt{A}]\!](R^\sharp, T^\sharp) \triangleq \begin{cases} \texttt{ff} & \text{if } R^\sharp = \bot_d^\sharp \\ \texttt{ff} & \text{else if } \texttt{isconst}^\sharp[\![\texttt{A}]\!]R^\sharp \\ \texttt{ff} & \text{else if } \texttt{A} = \texttt{n} \\ \texttt{ff} & \text{else if } \texttt{A} = \texttt{x} \wedge \texttt{x} \notin T^\sharp \\ \texttt{tt} & \text{else if } \texttt{A} = \texttt{x} \wedge \texttt{x} \in T^\sharp \\ \texttt{taint}^\sharp[\![\texttt{A}_1]\!](R^\sharp, T^\sharp) & \text{else if } \texttt{A} = \texttt{A}_1 \diamond \texttt{A}_2 \\ \quad \vee\, \texttt{taint}^\sharp[\![\texttt{A}_2]\!](R^\sharp, T^\sharp) & \end{cases} \tag{7.29}$$

The abstract semantics for `skip` statements is standard.

$$\mathcal{S}_t^\sharp[\![\texttt{skip}]\!](R^\sharp, E^\sharp, T^\sharp) \triangleq (R^\sharp, E^\sharp, T^\sharp) \tag{7.30}$$

As variables read from user input are the main sources of tainted data, we always taint variables read from input statements.

$$\mathcal{S}_t^\sharp[\![\texttt{x = input()}]\!](R^\sharp, E^\sharp, T^\sharp) \triangleq \textbf{let } (R_1^\sharp, E_1^\sharp) = \mathcal{S}_d^\sharp[\![\texttt{x = input()}]\!](R^\sharp, E^\sharp) \textbf{ in } (R_1^\sharp, E_1^\sharp, T^\sharp \cup \{\texttt{x}\}) \tag{7.31}$$

As observed in Section 7.7 (see Remark 7.2), random read statements can taint the assigned variable in case the user controls the position of the value which is read in the random input sequence. For the abstract semantics, it would be sound to always taint the assigned variable. Nevertheless, this is too coarse, and we propose an abstraction that improves the precision. The idea is to represent the sequence of random numbers as a queue: programs read from it at index `i`, and then increment `i`. In this model, `x = rand()` is syntactically substituted with `x = rand[i] ; i = i+1`. We assume that the abstract semantics $\mathcal{S}_d^\sharp[\![\mathtt{S}]\!]$ can handle reading from the queue. The special index variable `i` is then handled by the numeric domain as any other variable. We taint the result of `x = rand()` only if `i` is tainted: this happens when the user can control which number is read from the random sequence.

$$\begin{aligned}&\mathcal{S}_t^\sharp[\![\texttt{x = rand()}]\!](R^\sharp, E^\sharp, T^\sharp) \triangleq \\ &\quad \textbf{let } (R_1^\sharp, E_1^\sharp) = \mathcal{S}_d^\sharp[\![\texttt{x = rand[i] ; i = i+1}]\!](R^\sharp, E^\sharp) \textbf{ in} \\ &\quad \textbf{let } T_1^\sharp = \{\texttt{y} \in T^\sharp \mid \texttt{y} \neq \texttt{x}\} \cup \{\texttt{x} \mid \texttt{taint}^\sharp[\![\texttt{i}]\!](R^\sharp, T^\sharp)\} \textbf{ in} \\ &\quad (R_1^\sharp, E_1^\sharp, T_1^\sharp)\end{aligned} \tag{7.32}$$

Assignments can present runtime errors, so that we need to taint `ret` in case the user can trigger a program failure. To determine if there is an exploitable runtime error in the evaluation of an expression, we rely on the function $\texttt{exploit}^\sharp[\![\texttt{A}]\!] : (\mathbb{D}^\sharp \times \wp(\mathbb{V})) \to \mathbb{B}$. The function returns `tt` if there is a possible runtime error when evaluating `A`, and such an error can be triggered by the user. We assume the existence of a function $\texttt{zero}^\sharp[\![\texttt{A}]\!] : \mathbb{D}^\sharp \to \mathbb{B}$, which is provided by the numeric domain and returns `tt` if the evaluation of `A` is possibly zero. For instance, for intervals it can be implemented as follows:

$$\texttt{zero}_i^\sharp[\![\texttt{A}]\!]R^\sharp \triangleq [0,0] \sqsubseteq_i \mathcal{A}^\sharp_{\mathbb{V}_\mathbb{I}^\sharp}[\![\texttt{A}]\!]R^\sharp \tag{7.33}$$

By using $\texttt{taint}^\sharp[\![\texttt{A}]\!]$ and $\texttt{zero}^\sharp[\![\texttt{A}]\!]$, we can define $\texttt{exploit}^\sharp[\![\texttt{A}]\!]$ as follows.

$$\texttt{exploit}^\sharp[\![\texttt{n}]\!](R^\sharp, T^\sharp) \triangleq \texttt{ff} \tag{7.34}$$

$$\texttt{exploit}^\sharp[\![\texttt{x}]\!](R^\sharp, T^\sharp) \triangleq \texttt{ff} \tag{7.35}$$

$$\texttt{exploit}^\sharp[\![\texttt{A}_1 \diamond \texttt{A}_2]\!](R^\sharp, T^\sharp) \triangleq \begin{cases} \texttt{tt} & \text{if } \diamond = /,\ \texttt{zero}^\sharp[\![\texttt{A}_2]\!]R^\sharp,\ \texttt{taint}^\sharp[\![\texttt{A}_2]\!](R^\sharp, T^\sharp) \\ \texttt{tt} & \text{if } \texttt{exploit}^\sharp[\![\texttt{A}_1]\!](R^\sharp, T^\sharp) \text{ or } \texttt{exploit}^\sharp[\![\texttt{A}_2]\!](R^\sharp, T^\sharp) \\ \texttt{ff} & \text{otherwise} \end{cases} \tag{7.36}$$

Then, assignments taint the assigned variable in case evaluation of the arithmetic expression possibly depends on user input, and taint `ret` in case the user can trigger a runtime error.

$$\begin{aligned} &\mathcal{S}_t^\sharp[\![\texttt{x = A}]\!](R^\sharp, E^\sharp, T^\sharp) \triangleq \\ &\quad \textbf{let } (R_1^\sharp, E_1^\sharp) = \mathcal{S}_d^\sharp[\![\texttt{x = A}]\!](R^\sharp, E^\sharp) \textbf{ in} \\ &\quad \textbf{let } T_1^\sharp = \{\, \texttt{y} \in T^\sharp \mid \texttt{y} \neq \texttt{x} \,\} \cup \\ &\qquad \{\, \texttt{x} \mid \texttt{taint}^\sharp[\![\texttt{A}]\!](R^\sharp, T^\sharp) \,\} \cup \{\, \texttt{ret} \mid \texttt{exploit}^\sharp[\![\texttt{A}]\!](R^\sharp, T^\sharp) \,\} \textbf{ in} \\ &\quad (R_1^\sharp, E_1^\sharp, T_1^\sharp) \end{aligned} \tag{7.37}$$

The abstract semantics of statements composition is standard.

$$\mathcal{S}_t^\sharp[\![\texttt{S}_1\,;\,\texttt{S}_2]\!](R^\sharp, E^\sharp, T^\sharp) \triangleq \mathcal{S}_t^\sharp[\![\texttt{S}_2]\!](\mathcal{S}_t^\sharp[\![\texttt{S}_1]\!](R^\sharp, E^\sharp, T^\sharp)) \tag{7.38}$$

Example 7.8 (Safety-exploitability in assignments)
Consider the following program, analyzed with the interval domain:

```
x = input(); y = 1/x
```

After the execution of `x = input()` the variable `x` is tainted and can have any value. The statement `y = 1/x` taints `y` because $\texttt{taint}^\sharp[\![\texttt{1/x}]\!]$ returns `tt`, as `x` is

tainted in the input state. Furthermore, `ret` is tainted as well:

$$\begin{aligned}
&\texttt{exploit}^\sharp[\![\texttt{1/x}]\!](\{\texttt{x} \mapsto [-\infty, +\infty]\}, \{\texttt{x}\}) \\
=\ &\texttt{zero}^\sharp[\![\texttt{x}]\!]\{\texttt{x} \mapsto [-\infty, +\infty]\} \wedge \texttt{taint}^\sharp[\![\texttt{x}]\!](\{\texttt{x} \mapsto [-\infty, +\infty]\}, \{\texttt{x}\}) \\
=\ &[0, 0] \sqsubseteq_i [-\infty, +\infty] \wedge \texttt{x} \in \{\texttt{x}\} \\
=\ &\texttt{tt}
\end{aligned}$$

As discussed in Section 7.6 (see Example 7.4), if statements can generate *implicit flows* [126], namely dependencies that originate from the program control flow. When an attacker can control which branch of an if statement is executed, and in that branch a variable is assigned, then the variable could be tainted.

The set of variables that become tainted as a result of a tainted condition is traditionally overapproximated, when conditions are handled at all, with the variables that *syntactically* appear in the assignments of the branches (see Section 7.3). This is a coarse overapproximation, and we can improve this result by using the values of the variables. For instance, consider the following program:

```
x = y; if (y < x) { z = 10}
```

The assignment is never executed, and a relational analysis can deduce that `z` is never assigned. The traditional syntactic approach is not sufficient to infer this information. We rely on the function $\texttt{assigned}^\sharp[\![\texttt{S}]\!] : \mathbb{D}^\sharp \to \wp(\mathbb{V})$ that returns an overapproximation of the set of variables that are *semantically* assigned when executing `S`. If there is a state in the concretization of the abstract input $R^\sharp$ in which a variable `x` changes value during the execution of `S`, then $\texttt{x} \in \texttt{assigned}^\sharp[\![\texttt{S}]\!]R^\sharp$. Observe that in case an exploitable runtime error occurs, $\texttt{assigned}^\sharp[\![\texttt{S}]\!]R^\sharp$ includes `ret`, which does not syntactically appear in the program. A straightforward implementation can run the regular value analysis and inductively collect the variables that are assigned. While doing this, the function discards unreachable code and assignments that do not modify the state, such as `x = 0` when `x` is already 0, being effectively more precise than a syntactic approach. In Appendix C we report an implementation for $\texttt{assigned}^\sharp[\![\texttt{S}]\!]$ for the interval domain. We define the following

function to compute the set of variables that are tainted due to implicit flows.

$$\begin{aligned}&\texttt{diff}^\sharp[\![\texttt{if (B) } \texttt{S}_t \texttt{ else } \texttt{S}_e]\!](R^\sharp, T^\sharp) \triangleq \\ &\qquad \{\, \texttt{x} \in \texttt{assigned}^\sharp[\![\texttt{if (B) } \texttt{S}_t \texttt{ else } \texttt{S}_e]\!]R^\sharp \mid \texttt{taint}^\sharp[\![\texttt{B}]\!](R^\sharp, T^\sharp) \,\}\end{aligned} \tag{7.39}$$

Tainted variables can also become untainted due to conditionals. For instance, the variable x is not tainted inside of the then branch in the following program: `x = input(); if (x==0) {...}` (see Example 7.7). The reason is that x equals zero when entering the first branch, and constants are by definition not controlled by the user. Classic methods ignore this, and do not filter tainted variables after conditionals. This is sound, but we can again achieve better precision by taking into account the values of the variables. We define the function $\texttt{refine}^\sharp[\![\texttt{B}]\!] : (\mathbb{D}^\sharp \times \wp(\mathbb{V})) \to \wp(\mathbb{V})$ as follows:

$$\texttt{refine}^\sharp[\![\texttt{B}]\!](R^\sharp, T^\sharp) \triangleq T^\sharp \setminus \texttt{const}^\sharp(\texttt{test}^\sharp[\![\texttt{B}]\!]R^\sharp) \tag{7.40}$$

The function $\texttt{const}^\sharp$ is provided by the abstract domain, and returns the set of constant variables in the abstract state. For instance, in the interval domain it can be implemented as follows:

$$\begin{aligned}\texttt{const}_i^\sharp &: \mathbb{V}_\mathbb{I}^\sharp \to \wp(\mathbb{V}) \\ \texttt{const}_i^\sharp(R^\sharp) &\triangleq \{\, \texttt{x} \mid \texttt{isconst}_i^\sharp[\![\texttt{x}]\!]R^\sharp \,\}\end{aligned} \tag{7.41}$$

The function $\texttt{refine}^\sharp[\![\texttt{B}]\!]$ filters out the variables that are constant after the execution of the test B, improving the precision of the analysis. We can now give the definition of the abstract semantics for if statements.

$$\begin{aligned}&\mathcal{S}_t^\sharp[\![\texttt{if (B) } \texttt{S}_t \texttt{ else } \texttt{S}_e]\!](R^\sharp, E^\sharp, T^\sharp) \triangleq \\ &\quad \textbf{let } (R_t^\sharp, E_t^\sharp, T_t^\sharp) = \mathcal{S}_t^\sharp[\![\texttt{S}_t]\!](\texttt{test}^\sharp[\![\texttt{B}]\!]R^\sharp, E^\sharp, \texttt{refine}^\sharp[\![\texttt{B}]\!](R^\sharp, T^\sharp)) \textbf{ in} \\ &\quad \textbf{let } (R_e^\sharp, E_e^\sharp, T_e^\sharp) = \mathcal{S}_t^\sharp[\![\texttt{S}_e]\!](\texttt{test}^\sharp[\![\neg\texttt{B}]\!]R^\sharp, E^\sharp, \texttt{refine}^\sharp[\![\neg\texttt{B}]\!](R^\sharp, T^\sharp)) \textbf{ in} \\ &\quad \textbf{let } T_{te}^\sharp = \texttt{diff}^\sharp[\![\texttt{if (B) } \texttt{S}_t \texttt{ else } \texttt{S}_e]\!](R^\sharp, T^\sharp) \textbf{ in} \\ &\quad (R_t^\sharp \cup_d^\sharp R_e^\sharp, E_t^\sharp \cup_d^\sharp E_e^\sharp \cup_d^\sharp \texttt{err}^\sharp[\![\texttt{B}]\!]R^\sharp, T_t^\sharp \cup T_e^\sharp \cup T_{te}^\sharp)\end{aligned} \tag{7.42}$$

Example 7.9 (Abstract taint semantics and implicit flows)
The following program contains an implicit flow:

```
1 x = input();
2 if (x == 0) {
3   y = 1;
4 } else {
5   y = 2;
6 }
```

The abstract taint semantics infers that y is tainted at the end of the program, and this is due to the fact that y is assigned in a branch whose execution is influenced by a tainted variable:

$$
\begin{aligned}
&\texttt{diff}^\sharp[\![\texttt{if (x==0) y = 1 else y = 2}]\!](\{\texttt{x} \mapsto \top\}, \{\texttt{x}\}) \\
&= \{\, \texttt{z} \in \texttt{assigned}^\sharp[\![\texttt{if (x==0) y = 1 else y = 2}]\!]\{\texttt{x} \mapsto \top\} \mid \texttt{taint}^\sharp[\![\texttt{x==0}]\!](\{\texttt{x} \mapsto \top\}, \{\texttt{x}\}) \,\} \\
&= \{\, \texttt{z} \in \{\texttt{y}\} \mid \texttt{x} \in \{\texttt{x}\} \,\} \\
&= \{\texttt{y}\}
\end{aligned}
$$

Observe that x is not tainted inside of the then branch, because its value is constant. This can be observed, for instance, with an interval analysis:

$$
\begin{aligned}
&\texttt{refine}^\sharp[\![\texttt{x==0}]\!](\{\texttt{x} \mapsto [-\infty, +\infty]\}, \{\texttt{x}\}) \\
&= \{\texttt{x}\} \setminus \texttt{const}_i^\sharp(\texttt{test}^\sharp_{\mathbb{V}^\sharp_\mathbb{I}}[\![\texttt{x==0}]\!]\{\texttt{x} \mapsto [-\infty, +\infty]\}) \\
&= \{\texttt{x}\} \setminus \texttt{const}_i^\sharp(\{\texttt{x} \mapsto [0, 0]\}) \\
&= \{\texttt{x}\} \setminus \{\texttt{x}\} \\
&= \emptyset
\end{aligned}
$$

Example 7.10 (Abstract taint semantics with implicit flows and random reads)
The following program demonstrates various ways in which our analysis differs from other taint analyses.

```
1 x = input();
2 y = 1;
3 if (x == 0) {
```

```
4     z = rand();
5     if (z == 0) { 1/x }
6     if (z == 1) { y = z }
7   }
8   w = rand();
```

Firstly, we can infer that the program is exploitable: if the user inputs zero, then there is the possibility, depending on the sequence of random numbers, that a runtime error is triggered. The value analysis is important to infer that `x` is zero when performing the division, so that we can deduce that there is a division by zero. Secondly, we can use the semantic information inferred by the numeric domain to deduce that `y` is not tainted. Even if `y` is assigned inside of a branch that depends on the user's input, the variable `y` does not change, as it is still 1 after the execution of the statement. An interval analysis is sufficient to deduce this. Thirdly, we can infer that the variable `w` is tainted: depending on user input, it is assigned either to the first or the second value in the sequence of random numbers.

Further precision improvements can be implemented. For instance, consider the program `if (x < 10) { y = 0} else { y = 1}`. If the abstract value domain can determine that before the execution of the statement the value of `x` is less than 10, the statement is semantically equivalent to `y = 0`. This implies that, even if `x` is tainted, the user cannot control the value of `y`. When the analysis can infer that one of the two branches is never executed, the if statement can be substituted with the other branch, ignoring the implicit flows that are generated by the condition, and improving again the precision.

The abstract semantics for while statements is a classic limit computation that relies on the widening operator ∇ to guarantee convergence in a finite number of iterations. As the number of variables is finite, $\wp(\mathbb{V})$ has finite height, so that the widening operator for $\wp(\mathbb{V})$ is simply the set union. The abstract operator $(R_1^\sharp, E_1^\sharp, T_1^\sharp) \dot{\cup}^\sharp (R_2^\sharp, E_2^\sharp, T_2^\sharp)$ denotes $(R_1^\sharp \cup_d^\sharp R_2^\sharp, E_1^\sharp \cup_d^\sharp E_2^\sharp, T_1^\sharp \cup T_2^\sharp)$, and the operator $(R_1^\sharp, E_1^\sharp, T_1^\sharp) \dot{\nabla} (R_2^\sharp, E_2^\sharp, T_2^\sharp)$ denotes $(R_1^\sharp \nabla R_2^\sharp, E_1^\sharp \nabla E_2^\sharp, T_1^\sharp \cup T_2^\sharp)$.

$$\mathcal{S}_t^\sharp[\![\texttt{while (B) } \mathtt{S}_b]\!](R^\sharp, E^\sharp, T^\sharp) \triangleq \textbf{let } (R_f^\sharp, E_f^\sharp, T_f^\sharp) = \lim F^n(\bot_d^\sharp, \bot_d^\sharp, \emptyset) \textbf{ in} \tag{7.43}$$

$$(\texttt{test}^\sharp[\![\neg \mathtt{B}]\!]R^\sharp_f, E^\sharp_f, \texttt{refine}^\sharp[\![\neg \mathtt{B}]\!](R^\sharp_f, T^\sharp_f))$$

where $F(R^\sharp_1, E^\sharp_1, T^\sharp_1) \triangleq$ **let** $(R^\sharp_2, E^\sharp_2, T^\sharp_2) = \mathbb{S}^\sharp_t[\![\texttt{if (B) S}_b\texttt{ else skip}]\!](R^\sharp_1, E^\sharp_1, T^\sharp_1)$ **in**
$(R^\sharp_1, E^\sharp_1, T^\sharp_1) \dot{\nabla} ((R^\sharp, E^\sharp, T^\sharp) \dot{\cup}^\sharp (R^\sharp_2, E^\sharp_2, T^\sharp_2))$

Remark 7.4 (Precision improvement due to value-taint collaboration)
In principle, it is possible to first run a value analysis, and then use the inferred numeric invariants in a taint analysis to prove safety-nonexploitability. Nevertheless, as shown by the following program, executing the two together achieves strictly superior precision.

```
1 x = input();
2 if (x <= 0) {
3   x = 1;
4 }
5 while (tt) {
6   1 / x;
7   x = rand();
8 }
```

The invariant inferred at line 6 entails that `x` can be zero, so that there is a potential runtime error. Furthermore, a taint analysis infers that `x` is tainted at the same program location. By combining this information, the division by zero is exploitable. However, if we execute the value and the taint analyses together, we can observe that it is never true *at the same time* that `x` is 0 and tainted, so that the program failure cannot be triggered by an attacker. Our framework runs the two analyses together, and is thus able to prove that the program is safety-nonexploitable.

In this section, we defined a value-sensitive semantic taint analysis that can prove safety-nonexploitability. In Chapter 8, we implement our analysis, and we evaluate its precision and performance compared to a classic safety analysis. Our experiments show that we are able to filter out more than 70% of the alarms raised by the normal analyzer in real-world programs.

7.9. Related work

In this section, we discuss related work. In particular, we describe secure information flow, hyperproperties verification, security properties verification by abstract interpretation, slicing, and error classification techniques.

7.9.1. Secure information flow

In [126] the authors propose the first mechanism to verify the secure flow of information in a program, namely checking that a program cannot cause supposedly nonconfidential results to depend on confidential input data. Their formulation of the problem is based on the syntax of a program, and does not take into account its semantics. The concept of secure information flow is related to *noninterference* [123, 124, 125], which is a semantic definition. A program is *noninterferent* if its public output data does not depend on private input data. Checking that a program cannot cause nonconfidential results to depend on confidential input data has been widely studied through type systems [220, 221, 222, 223, 224, 225, 226, 227, 228, 229, 230]. Because these works perform only syntactic checks without taking into account semantic information, the results are generally imprecise. For instance, let $\mathtt{x}_{\mathtt{priv}}$ be a secret variable, and $\mathtt{x}_{\mathtt{pub}}$ be a public variable. The program $\mathtt{x}_{\mathtt{pub}} = \mathtt{x}_{\mathtt{priv}}\,;\, \mathtt{x}_{\mathtt{pub}} = 0$ would be labelled as interferent by type systems-based approaches, which are not precise enough to infer that the value of the public variable does not actually depend on the secret one. In [231], the authors put forward an information flow static analysis using a Hoare-like logic to achieve superior precision. The analysis is sound and draws ideas from abstract interpretation theory to compute independencies between variables. In contrast to type systems-based techniques and [231], our approach tracks the flow of data generated by the user through a semantic taint analysis, which achieves enhanced precision by leveraging an overapproximation of the values of the variables.

Noninterference and safety-nonexploitabilty are closely related: nonexploitability can be seen as a variation of noninterference where the only public output variable is `ret`. Nevertheless, we do not rely on the static partitioning of variables into public and private, as our definition supports dynamic user input reads. Our framework can be used to prove noninterference: it is sufficient to read all private

input variables at the beginning of the program, and then verify that the public output variables are not tainted. On the contrary, traditional methods to prove noninterference cannot prove safety-nonexploitability, as they do not take the values of the variables into account.

The framework of *abstract noninterference* [232, 233, 234, 235, 236, 237, 238] is a generalization of classic noninterference parametric on abstractions modelling different aspects of the information flow. The *observer abstraction* and the *maximal input variation abstraction* are part of the attacker model, and respectively abstract what can be observed in the output by an attacker and what she can control/learn from the input. The *selection abstraction* is used to select for which inputs we check abstract noninterference. Safety-nonexploitability can be regarded as an instance of abstract noninterference where the observer abstraction selects from the output only the value of the return variable, the maximal input variation abstraction is simply the identity function, and the selection abstraction checks safety-nonexploitability only when pairs of inputs (in our framework, initial states) agree on everything but the user input.

In [239] the authors put forward a technique to prove the absence of unwanted accesses to objects within the context of Java Card applications [240]. Similarly to our approach, their technique relies on an overapproximating value analysis to prove a security property. On the other hand, proving the absence of unwanted accesses to objects is a safety property, while safety-nonexploitability is a hypersafety property. The analysis that the authors propose in [239] collects a set of constraints that describe the flow and the values of the data, and these constraints can be solved using a fixpoint algorithm. The analysis they perform is an *information flow analysis*, even though they do not rely directly on taint or dependency analysis. Furthermore, in [239] implicit flows are not addressed. Other flow analyses for Java Card applications are described in [241, 242].

7.9.2. Hyperproperties verification

Clarkson and Schneider [50] put forward the framework of *hyperproperties*, namely program properties that relate different sets of executions. Hyperproperties are able to express security policies, such as secure information flow. K-hypersafety properties [50] can be verified with traditional techniques for safety properties on

the k-times self-composed system [243, 244], even though this can be computationally expensive [245]. HyperLTL and HyperCTL/CTL* [246, 247] define extensions of temporal logic able to quantify over multiple traces to address the verification of hyperproperties. There are a number of model checking-based techniques for hyperlogic verification [247, 248, 249, 246, 250]. MCHyper [246], AutoHyper [251], and HyperQube [250] are model checkers that support the verification of hyperproperties.

7.9.3. Security properties verification by abstract interpretation

Cousot [252] put forward a semantic definition of dependencies in the abstract interpretation framework. He proposes a sound analysis of dependencies, capable of proving noninterference. Similarly to us, he does not rely on hypersemantics, using standard abstract interpretation techniques. Nevertheless, the abstract dependency semantics is not structural (i.e., defined by induction on the program syntax), as it does not take the values of variables into account. The author proposes leveraging the values of the variables to give a structural definition of the semantics, and this work attempts to implement such an extension. Since his definition of the dependency semantics does not take into account the values of the variables, it is not possible to define an analysis that leverages numeric abstract domains to enhance the precision of the dependency analysis. Another significant difference is that the dependencies are relative to the *initial values* of variables. Our analysis computes *dynamic tainting*, which is the dependency of a variable from *any* input statement (including those within conditionals and loops), so that it generalizes the dependency analysis from the beginning of the program.

There are numerous papers that use an alternative version of the abstract interpretation framework based on *hypersemantics* [219, 215, 216, 217, 218], where the concrete domain is a set of sets of states, rather than a set of states. This is to overcome the difficulties related to the fact that not every hyperproperty is subset-closed, and classic overapproximation techniques seem to fail. However, as argued in this manuscript and [252], this is not the case for standard noninterference and safety-nonexploitability. Relying on the classic abstract interpretation framework allows using the large library of existing abstract domains, and leveraging the semantic information inferred by such domains is not only essential

for safety-nonexploitabilty, but also enhances the precision of the taint analysis. Another approach to noninterference verification is introduced in [253], where the authors combine abstract interpretation with symbolic execution to define a sound analysis.

Deng and Cousot propose a semantic definition of *responsibility* by abstract interpretation [254, 255, 256]. Informally, an entity is *responsible* for an observable behaviour in case it is free to choose its input value, and such a choice is the first one that *ensures* the occurrence of the behaviour in the execution. We believe it is possible to instantiate responsibility to runtime errors in order to identify the exact statement that is responsible for a failure to occur. Our definition of safety-nonexploitability is less precise than responsibility, as we abstract away which input read statements are involved in triggering a runtime failure. Even if we tracked this information, our input-output semantic model is not precise enough to capture exactly which statement is the *first* to *ensure* a bug to happen, for which we would need a trace semantics. Nevertheless, this precision loss makes our definition more appropriate to prove that there exists no input statement that might be responsible for the occurrence of any runtime error.

Infer [214], Pysa [201], and Julia [194] are static analyzers based on abstract interpretation that support taint analysis. All these tools do not detect *implicit flows*, being effectively unsound in our framework. The analyzers can track taint information, but they cannot classify runtime errors as exploitable or not. Frama-C [187] with the Eva plugin [186] supports an experimental taint analysis for C which is capable of classifying variables as possibly user-controlled or not. The analysis is described in the Eva user manual [257, Section 6.7.8], which is, at the time we write, at version 28.0. The taint analysis requires to annotate functions with taint source information, which is then propagated and checked against taint verification assertions. Similarly to our work, Eva supports both implicit and explicit flows. In our experiments with Eva, we observed that the taint analysis does not perform advanced reductions with the numeric domain such as those that we implement in our framework. Since Frama-C with the Eva plugin has both a taint analysis and a value analysis, there are all the components necessary to implement a safety-nonexploitability analysis.

7.9.4. Slicing

Program slicing [258, 259, 260, 261], initially introduced by Mark Weiser in [258], is a technique to statically determine a set of statements that may affect the values of the variables at a specific program point. Slicing has been used in different areas such as debugging [258], software maintenance [262], comprehension [263, 264], and re-engineering [265]. The slicing can be *syntactic* [258, 266] if it is based only on the syntax of programs, or *semantic* [267, 268, 269, 270, 271, 272, 273] if it also considers the semantic behaviour.

Slicing is usually *backwards,* meaning that the interest is on the part of the program that affects an observation associated with a slicing criterion. A significant difference with our work is that we perform a *forwards* analysis. In terms of slicing, it can be thought as computing the set of program points with a runtime error that depend on a user input statement. In this sense, our work is more closely related to *forward slicing techniques* [268], where the interest is on the portion of the program that is affected by a particular statement. Another difference with slicing techniques is that they are focused on the program points that are impacted by (or impact) a *single statement*. In our analysis, we are concerned with computing the dependencies from *all* user input statements to *all* program statements that present a runtime error.

7.9.5. Errors classification

In [213] the authors put forward the concept of *robust reachability*. A runtime error is robustly reachable if a controlled input can make it so the bug is reached whatever the value of uncontrolled inputs. The authors use symbolic execution and bounded model checking techniques to find robustly reachable bugs. Similarly to this work, [213] classifies runtime failures by their dangerousness and filters out less interesting alarms that do not concern security issues. Nevertheless, the concept of robustly reachable runtime error is different from safety-nonexploitability: a bug is considered robustly reachable even if it is triggered *for all* possible user input, while such an error is not exploitable according to our formal definition of exploitability. In fact, we require the user input to be actually involved in triggering an error to consider a program exploitable.

While robust reachability ensures the perfect reproducibility of a bug, it fails

to report errors that are *mostly* reproducible, namely errors that can be triggered for the large majority of uncontrolled inputs but not all. To address this limitation, the authors extend the framework to support the generation of constraints on the uncontrolled inputs that ensure a target error to be robustly reached [274]. The technique is useful to explain the conditions under which a certain runtime failure can always be triggered by an attacker. In future work, we would like to apply the ideas proposed in [274] to safety-nonexploitability in order to put forward a more intelligible analysis. In particular, it would be interesting to report under which conditions an exploitable runtime error can be triggered.

Other techniques relying on probability theory to differentiate classes of bugs have been proposed. They include *probabilistic model checking* [275, 276], *probabilistic abstract interpretation* [277, 278, 279, 280], *quantitative robust reachability* [281], and *quantitative information flow analysis* [282]. An interesting extension of this work would be to use ideas from these approaches to put forward a *quantitative* exploitability analysis to classify more finely the level of threat caused by alarms.

7.10. Conclusion

In this chapter, we introduced the novel definition of safety-nonexploitability, which we leveraged to put forward a sound analysis by abstract interpretation. The framework supports constructs that are essential to analyze real-world programs, such as nondeterminism and dynamic user input reads. Our analysis performs a semantic taint analysis that achieves superior precision through a modular reduction with existing numeric abstract domains. The theoretical framework bridges the gap between traditional safety properties and security hyperproperties, and our analysis can rule out the existence of exploitable runtime errors in programs. In Chapter 8 we implement our analysis in the Mopsa-Nexp tool, which we use to evaluate the effectiveness of our technique.

In future work, we would like to extend our analysis to prove the absence of other classes of exploitable bugs. A promising path forward is to leverage probability theory to perform a *quantitative* exploitability analysis capable of further reducing the number of alarms. Another interesting extension of this work is to adapt our framework to rule out the existence of exploitable liveness errors, such

as exploitable livelocks in multithreaded programs. Furthermore, in future work we would like to enhance the analysis output to report the conditions under which each vulnerability can be triggered. Implementing this improvement would require extending our framework to support the generation of exploitability constraints, similarly to [274]. We believe that enhancing our analysis with explanations about the origin of warnings would make it more intelligible.

Chapter 8

Safety Nonexploitability Experimental Evaluation

We implemented and evaluated the first analyzer for safety-nonexploitability in the MOPSA [48] static analysis platform. The analysis targets a large subset of C and it is *fully automatic*. We analyzed 77 real-world programs, each up to 4,188 lines long, taken from the GNU Coreutils package, to which we added 13,261 test cases taken from the Juliet test suite developed by NIST [49]. We found that our tool can *prove* that more than 70% of the warnings previously raised by the analyzer (3,498 over 4,715) cannot be triggered by an attacker, while incurring a performance overhead of less than 16%. In Section 8.1 we describe some details of our implementation, while in Section 8.2 we present our benchmarks and our experimental evaluation, which we further analyze in Section 8.3.

8.1. Implementation

We propose MOPSA-NEXP, the *first* analyzer dedicated to safety-nonexploitability. We implemented our analysis for a large subset of C in the MOPSA framework [48], which is a modular platform to build static analyzers based on abstract interpretation. MOPSA offers an extensive collection of ready-to-use abstract domains for analyzing C and Python, providing the flexibility to tune the tradeoff between precision and performance. MOPSA is implemented in 120,000 lines of OCaml

code, and our safety-nonexploitability analysis accounts for around 10,000 of them. Thanks to MOPSA's modular design, we were able to use most of the C analysis with minimal modifications.

In our implementation, we maintain taint information at the level of *memory blocks*, i.e. we perform a *field-insensitive* taint analysis. While this can result in a loss of precision, the implementation is simple and efficient. Proposing an enhanced field-sensitive taint analysis for C is out of the scope of this manuscript, and it is left as future work. As MOPSA performs dynamic expression rewriting to encourage a design based on layered semantics, to retrieve sources of tainted data, during the analysis we have to consider the expressions' rewriting history.

Our analysis can detect a wide variety of runtime errors, including double frees, index-out-of-bounds, and null pointer dereferences. While the formal presentation in this article, for the sake of simplicity, only supports division-by-zero errors, it was trivial to adapt our analysis to identify different types of failures. In the report of the analyzer each warning is classified as possibly exploitable or not, and we infer a sound overapproximation of both the regular runtime errors and the exploitable ones. All the warnings that are not labelled as exploitable are thus *proved* to be nonexploitable. If the analyzer does not report *any* exploitable warning, then this is a proof that the program is safety-nonexploitable. Our analysis can detect the following runtime errors:

- Memory: null pointer dereference, invalid pointer dereference, index out-of-bounds, dangling pointer dereference, use after free, double free, modification of read-only memory.
- Integer arithmetic: division by zero, integer overflow, invalid bit shift operation, invalid pointer comparison, invalid pointer subtraction.
- Floating-point arithmetic: floating point division by zero, floating point invalid operation, floating point overflow.
- Variadic arguments: insufficient number of variadic arguments, insufficient number of format arguments, invalid format argument type.
- Violation of a stub language contract (see [283] for more information about the stub modeling language)

TABLE 8.1. List of C functions that generate tainted data in MOPSA-NEXP

`getchar`	`getc_unlocked`	`getline`	`recvfrom`
`getchar_unlocked`	`getw`	`getdelim`	`scanf`
`fgetc`	`fgets`	`fread`	`fscanf`
`fgetc_unlocked`	`fgets_unlocked`	`fread_unlocked`	`sscanf`
`getc`	`gets`	`recv`	

The functions that read data from the user are part of the C standard library. They include, for instance, `getchar`, `scanf`, and `recv`. MOPSA provides a stub modelling language to specify the behaviour of library functions [283]. We have extended this language to support the fact that some functions generate tainted data, and then we annotated our stubs for the C standard library to take into account the taint information. In Table 8.1 we report the complete list of functions that generate tainted data in our analysis. Differently from existing taint analysis techniques, in our approach we do not have to explicitly annotate the sinks (i.e., the program locations where user-controlled data is forbidden to flow), as they are not statically known and correspond to possible runtime errors. For this reason, the user of the analyzer does not have to annotate her source code to run the nonexploitability analysis, which is *fully automatic*.

8.2. Performance and precision evaluation

To assess the usefulness of our tool, we have analyzed real-world C programs from the GNU Coreutils package, which is a collection of command-line utilities. The test suite is composed of 77 programs that are long up to 4,188 lines of code each. To them, we added a large set of short C programs taken from the Juliet test suite developed by NIST [49]. These programs contain examples of various runtime errors that can trigger well-known security vulnerabilities. In fact, Juliet is based on the CWE database [284], which enumerates vulnerabilities and focusses on security. The tested runtime errors include double frees, index out-of-bounds, and null pointer dereferences. The test cases are specifically designed to assess the precision of static analysis tools, and use a large set of features from the C standard. For Juliet, we considered 13,261 different test cases that amount to a total of 2,861,980 lines of code. Each test case comes with two versions: one that

TABLE 8.2. Safety-nonexploitability evaluation results

Test suite	Domain	Analyzer	Alarms		Time	
Coreutils	Intervals	MOPSA	4,715		1:17:06	
		MOPSA-NEXP	1,217	(-74.19%)	1:28:42	(+15.05%)
	Octagons	MOPSA	4,673		2:22:29	
		MOPSA-NEXP	1,209	(-74.13%)	2:43:06	(+14.47%)
	Polyhedra	MOPSA	4,651		2:12:21	
		MOPSA-NEXP	1,193	(-74.35%)	2:30:44	(+13.89%)
Juliet	Intervals	MOPSA	49,957		11:32:24	
		MOPSA-NEXP	13,906	(-72.16%)	11:48:51	(+2.38%)
	Octagons	MOPSA	48,256		13:15:29	
		MOPSA-NEXP	13,631	(-71.75%)	13:41:47	(+3.31%)
	Polyhedra	MOPSA	48,256		12:54:21	
		MOPSA-NEXP	13,631	(-71.75%)	13:21:26	(+3.50%)

triggers a runtime failure, and one where the error is fixed. We run our analysis on both versions. An artifact to reproduce our experimental evaluation is available on Zenodo [54].

We compare the performance and number of alarms between MOPSA-NEXP and MOPSA. The analyses are parametric in the underlying abstract numeric domain, and we consider intervals, octagons [158], and polyhedra. Observe that to compare only the number of alarms raised by the two analyzers it is not necessary to run both tools, as MOPSA-NEXP can report all warnings raised by MOPSA. Notice that while the ground truth about the errors provided with Juliet can be used to evaluate the precision of a classic safety analysis, this is not the case for safety-nonexploitability. In fact, the benchmarks categorize the test cases as either dangerous or not, but they do not include any information about whether an attacker can trigger the errors. We ran our experiments on a server with 128GB of RAM, with 48 Intel Xeon CPUs E5-2650 v4 @ 2.20GHz and Ubuntu 18.04.5 LTS. In Table 8.2 we report the results of our experiments.

For Coreutils, in the case of intervals, our analysis was able to *prove* that 3,498 over 4,715 runtime errors previously reported by the analyzer cannot be triggered by an attacker. For octagons and polyhedra, our analysis proved that respectively 3,464 and 3,458 potential runtime errors over 4,673 and 4,651 are not exploitable.

```c
int i;
i = rand();
int buf[10];
if (i>=0) {
  buf[i] = 1;
}
```

A. Safety-nonexploitable C program

```c
int i;
scanf("%d", &i);
int buf[10];
if (i>=0) {
  buf[i] = 1;
}
```

B. Safety-exploitable C program

```c
int i;
scanf("%d", &i);
int buf[10];
if (i>=0&&i<10){
  buf[i] = 1;
}
```

C. Safety-nonexploitable C program due to index sanitization

FIGURE 8.1. Simplified versions of test cases for index out-of-bounds

Overall, this results in filtering out 74.13%-74.35% of the warnings. We found similar results for Juliet, where MOPSA-NEXP was able to prove that 71.75%-72.16% of the warnings are not exploitable. For Coreutils, MOPSA-NEXP raises 1,193 to 1,217 warnings, which are those that can be potentially triggered by an attacker. The user of the analyzer could prioritize those alarms over the regular ones, as they are comparatively more dangerous.

The exploitability analysis incurs a performance overhead ranging from 13.89% to 15.05% for Coreutils and 2.38% to 3.5% for Juliet. During the analysis we consider expressions' rewriting history to preserve taint information, and this history is sensibly larger in real-world programs, which justifies the performance overhead difference between Coreutils and Juliet. Observe that we found octagons to be less efficient than polyhedra. This is due to the fact that MOPSA relies on the APRON [200] library, which uses a sparse representation for polyhedra, and can be very efficient if the number of variables is low and there are few constraints. As octagons use a dense representation, even if their algorithmic complexity is better, they are slightly slower in our case.

Figures 8.1.A to C represent simplified versions of test cases that trigger index out-of-bounds failures in the Juliet benchmarks. The program in Figure 8.1.A contains a potential runtime error. Nevertheless, the failure cannot be triggered by an attacker, so that the program is safety-nonexploitable. In Figure 8.1.B, the program becomes safety-exploitable as the user has control over the buffer's read index and can potentially trigger the index out-of-bounds error. In the last program represented in Figure 8.1.C, even though the user controls the index `i`, proper sanitization of `i` prevents the program from incurring in a (safety-exploitable)

runtime error. Observe that the values of the variables are essential to prove safety-nonexploitability in the last case. MOPSA-NEXP correctly classifies the programs respectively as nonexploitable, exploitable, and nonexploitable.

8.3. Discussion

We observed that MOPSA-NEXP is able to consistently filter out more than 70% of the warnings raised by the regular analyzer, while imposing low performance overhead. The Juliet test cases show that MOPSA-NEXP can handle almost the whole C specification, while the Coreutils experiments confirm that our analysis is effective even for real-world programs. The significant advantage of being able to classify each warning as security-critical or not outweighs the reasonable performance cost overhead. Observe that the alarms raised by MOPSA-NEXP are a subset of those reported by MOPSA. This implies that the exploitability analysis is, in the worst case, as precise as the regular analysis.

While it would be desirable to determine how many truly exploitable alarms are raised by MOPSA-NEXP, this cannot be done automatically. In fact, there is no ground truth that classifies program errors as nonexploitable or not, so that human inspection is the only option. In future work, we would like to conduct such an inspection.

8.4. Conclusion

We implemented our analysis in the MOPSA-NEXP tool, the first analyzer dedicated to safety-nonexploitability. The tool is fully automatic, and to assess its effectiveness, we evaluated it on a large set of real-world C programs. The analyzer can consistently *prove* that more than 70% of the previously raised warnings cannot be triggered by an attacker, all while incurring less than 16% performance overhead. While usually the number of false positives is lowered by increasing the precision of the abstract domains, we take an orthogonal approach by reporting only the alarms that can be triggered by an attacker. By leveraging the fundamental observation that security-related warnings are more dangerous than the others, our technique dramatically reduces the noise generated by false alarms, enhancing

the usefulness of the analyzer.

Part IV

Conclusion & Future Work

Chapter 9

Conclusion & Future Work

This thesis aims at developing and implementing formal techniques that can prove the absence of security-related vulnerabilities in software systems. We focused our attention on two notable cases: Regular Expression Denial of Service attacks (ReDoS), and exploitable runtime errors. For each case, we precisely characterized the semantics of the systems that we were considering, which allowed us to formally define the property we were interested in proving. The analyses we proposed are *automatic* and *sound*, meaning that they can rule out of existence security vulnerabilities in software systems without user intervention. Our experiments on real-world data give empirical evidence of the effectiveness of the techniques that we put forward.

For ReDoS, we started by giving a formal characterization of the matching engines' behaviour in terms of matching trees, which is, to the best of our knowledge, the first of its kind. By relying on such a definition, we were able to precisely define ReDoS attacks in terms of matching trees' sizes. This characterization allowed us to formally reason on ReDoS without having to resort to automata. We also defined a sound analysis to extract an overapproximation of the language of words that can trigger exponential matching. Our tool, RAT, demonstrated the effectiveness of our approach on a large dataset of 74,669 regular expressions. In fact, RAT is faster–often by orders of magnitude–than most other ReDoS detectors. Furthermore, RAT can report an overapproximation of the language of dangerous words, being strictly more expressive than most other tools. While raising a low number of false positives, RAT *is the only ReDoS detector that does not report false negatives*.

In the case of exploitable runtime errors, we considered a concrete semantics that supports features such as nondeterminism and dynamic user input reads. We put forward the novel property of safety-nonexploitability, which formalizes the idea that users cannot interfere with the correctness of the program. By giving an alternative characterization in terms of semantically tainted (i.e., user-controlled) variables, we showed that safety-nonexploitability can be proved by relying on a taint analysis. We defined a sound analysis by abstract interpretation that combines taint analysis with a classic value analysis. The numeric domain detects the runtime errors, while the taint analysis labels those errors as exploitable or not. Moreover, the combination of the two domains results in a strictly more precise taint analysis. Our experiments show that *our technique is able to consistently prove that more than 70% of the alarms raised by the regular analyzer on real-world software are not exploitable*.

We believe that sound tools capable of automatically proving the absence of security breaches are extremely valuable, particularly in a scenario where software systems are becoming increasingly complex. Proving that a program is *secure* is challenging, especially because there is no general, formally defined characterization of *secure programs*. Security is rather a set of complementary definitions, with new ones regularly being proposed by researchers. We hope that this work provides a new innovative perspective on security through formal reasoning and represents a step forward towards more secure software systems.

For future work, there are numerous potential directions we would like to explore.

Support for regular expression advanced features. Even if only a small portion of the regular expressions in real-world programs use advanced features such as backreferences and lookarounds, it would be interesting to extend our ReDoS analysis in order to analyze them. This task requires enhancing our semantic framework to natively support those features. Then, sound abstractions would make it possible to prove the absence of ReDoS vulnerabilities in the whole dataset of regular expressions that we considered in our experiments.

ReDoS superlinear vulnerabilities. Attackers that exploit superlinear but not exponential matching must be allowed to insert very large strings. For this

reason, superlinear vulnerabilities are sensibly less dangerous than exponential ones. Nevertheless, when the exponent of the polynomial is high, the matching can be time-consuming. While in Section 4.5 we proposed some ideas about how to implement the ReDoS superlinear analysis, it is still not clear how to determine the exponent of the polynomial, which is crucial to report meaningful results to the users.

Combining ReDoS and program analyis. Our ReDoS detection framework analyzes regular expressions in isolation. It would be interesting to combine RAT with a program analysis in order to prove the absence of ReDoS vulnerabilities in the context of vulnerable programming languages such as Python or Javascript. A promising idea is to combine a string analysis with our ReDoS detection framework. When the analyzer encounters a regular expression matching, it intersects the language of dangerous words with the possible strings that can be matched. If this intersection is empty, the matching is proved to be safe.

Extending nonexploitability to other software faults. Our definition of nonexploitability classifies a program as secure when the user cannot exploit a runtime error. It would be interesting to extend this definition to other types of software faults. One example is the absence of livelocks in multithreaded programs that can be triggered by an attacker. There is a long list of software vulnerabilities that can be exploited by a malicious user, and in our work we just scratched the surface of the possibilities that the nonexploitability framework could offer.

Enchancing the report of the analyzer. Currently, our analysis labels warnings as either security-critical or not. A promising research direction is to enhance the analysis output to report the conditions under which each vulnerability can be triggered. Implementing this improvement would require extending our framework to support the generation of exploitability constraints, similar to what was done by Sellami et al. [274]. We believe that enhancing our analysis with explanations about the origin of warnings would make the analyzer more intelligible.

Combining ReDoS and nonexploitability. A promising research direction is combining ReDoS and nonexploitability analyses. A program would be classified as ReDoS-nonexploitable if there are no ReDoS vulnerabilities that can be triggered by users. This analysis is best suited to target vulnerable programming languages such as Python and Javascript, and a string analysis to overapproximate the words that are matched in regular expressions would increase the precision of the analysis. We believe that by pairing the nonexploitability framework with our ReDoS analysis, it would be possible to prove the absence of exploitable ReDoS attacks in real-world programs.

Bibliography

[1] Dominique Brière and Pascal Traverse. AIRBUS A320/A330/A340 electrical flight controls: A family of fault-tolerant systems. In *Digest of Papers: FTCS-23, The Twenty-Third Annual International Symposium on Fault-Tolerant Computing, Toulouse, France, June 22-24, 1993*, pages 616–623. IEEE Computer Society, 1993. doi:10.1109/FTCS.1993.627364.

[2] Daniel Kästner, Jörg Barrho, Ulrich Wünsche, Marc Schlickling, Bernhard Schommer, Michael Schmidt, Christian Ferdinand, Xavier Leroy, and Sandrine Blazy. Compcert: Practical experience on integrating and qualifying a formally verified optimizing compiler. In *ERTS2 2018-9th European Congress Embedded Real-Time Software and Systems*, pages 1–9, 2018.

[3] Joel Finch. Toyota sudden acceleration: a case study of the national highway traffic safety administration-recalls for change. *Loy. Consumer L. Rev.*, 22:472, 2009.

[4] Complete guide to gdpr compliance in the European Union, 2024. URL: https://gdpr.eu/.

[5] Patrick Cousot. *Principles of Abstract Interpretation*. The MIT Press, 2022. URL: https://mitpress.mit.edu/9780262044905/principles-of-abstract-interpretation.

[6] W. Eric Wong, Xue-Lin Li, and Phillip A. Laplante. Be more familiar with our enemies and pave the way forward: A review of the roles bugs played in software failures. *J. Syst. Softw.*, 133:68–94, 2017. doi:10.1016/J.JSS.2017.06.069.

[7] Herb Krasner. The cost of poor software quality in the us: A 2020 report. *Proc. Consortium Inf. Softw. QualityTM (CISQTM)*, pages 1–46, 2021.

[8] Robert Skeel. Roundoff error and the patriot missile. *SIAM News*, 25(4):11, 1992.

[9] Mark Dowson. The ariane 5 software failure. *ACM SIGSOFT Softw. Eng. Notes*, 22(2):84, 1997. doi:10.1145/251880.251992.

[10] The ChatGPT converstational agent, 2024. URL: https://chat.openai.com/.

[11] The Gemini converstational agent, 2024. URL: https://gemini.google.com/app.

[12] The Github Copilot AI-powered coding assistant, 2024. URL: https://marketplace.visualstudio.com/items?itemName=GitHub.copilot.

[13] Microsoft has over a million paying Github Copilot users: CEO Nadella, 2024. URL: https://www.zdnet.com/article/microsoft-has-over-a-million-paying-github-copilot-users-ceo-nadella/.

[14] Hammond Pearce, Baleegh Ahmad, Benjamin Tan, Brendan Dolan-Gavitt, and Ramesh Karri. Asleep at the keyboard? assessing the security of GitHub Copilot's code contributions. In *Security and Privacy, SP*, pages 754–768. IEEE, 2022. doi:10.1109/SP46214.2022.9833571.

[15] Owura Asare, Meiyappan Nagappan, and N. Asokan. Is GitHub's Copilot as bad as humans at introducing vulnerabilities in code? *Empir. Softw. Eng.*, 28(6):129, 2023. doi:10.1007/S10664-023-10380-1.

[16] Yujia Fu, Peng Liang, Amjed Tahir, Zengyang Li, Mojtaba Shahin, and Jiaxin Yu. Security weaknesses of Copilot generated code in GitHub. *CoRR*, abs/2310.02059, 2023. arXiv:2310.02059, doi:10.48550/ARXIV.2310.02059.

[17] Arghavan Moradi Dakhel, Vahid Majdinasab, Amin Nikanjam, Foutse Khomh, Michel C. Desmarais, and Zhen Ming (Jack) Jiang. GitHub Copilot AI pair programmer: Asset or liability? *J. Syst. Softw.*, 203:111734, 2023. doi:10.1016/J.JSS.2023.111734.

[18] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976. URL: https://www.worldcat.org/oclc/01958445.

[19] The Scala programming language, 2024. URL: https://www.scala-lang.org/.

[20] The Rust programming language, 2024. URL: https://www.rust-lang.org/.

[21] The Swift programming language, 2024. URL: https://www.swift.org/.

[22] Henry Gordon Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical society*, 74(2):358–366, 1953.

[23] Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Comput. Surv.*, 51(3):50:1–50:39, 2018. doi:10.1145/3182657.

[24] Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, 1986. doi:10.1145/5397.5399.

[25] Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems, TACAS*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004. doi:10.1007/978-3-540-24730-2_15.

[26] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. *Principles of Programming Languages, POPL*, 1977.

[27] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. A static analyzer for large safety-critical software. In *Programming Language Design and Implementation, PLDI*, pages 196–207. ACM, 2003. doi:10.1145/781131.781153.

[28] Alain Ourghanlian. Evaluation of static analysis tools used to assess software important to nuclear power plant safety. *Nuclear Engineering and Technology*, 47(2):212–218, 2015.

[29] Microsoft: A proactive approach to more secure code, 2019. Accessed: 2023-08-30. URL: https://msrc.microsoft.com/blog/2019/07/a-proactive-approach-to-more-secure-code/.

[30] Munirul Ula, Zuraini Ismail, and Zailani Mohamed Sidek. A framework for the governance of information security in banking system. *Journal of Information Assurance & Cyber Security*, 2011:1–12, 2011.

[31] Kjell Jørgen Hole, Vebjørn Moen, and Thomas Tjøstheim. Case study: Online banking security. *IEEE Secur. Priv.*, 4(2):14–20, 2006. doi:10.1109/MSP.2006.36.

[32] Vishal R Ambhire and Prakash S Teltumde. Information security in banking and financial industry. *International Journal of Computational Engineering & Management*, 14, 2011.

[33] Matt Bishop. About penetration testing. *IEEE Secur. Priv.*, 5(6):84–87, 2007. doi:10.1109/MSP.2007.159.

[34] Sugandh Shah and Babu M. Mehtre. An overview of vulnerability assessment and penetration testing techniques. *J. Comput. Virol. Hacking Tech.*, 11(1):27–49, 2015. doi:10.1007/S11416-014-0231-X.

[35] Scott A. Crosby and Dan S. Wallach. Denial of service via algorithmic complexity attacks. In *USENIX Security Symposium*. USENIX Association, 2003. doi:10.1007/11506881_10.

[36] Cristian-Alexandru Staicu and Michael Pradel. Freezing the web: A study of ReDoS vulnerabilities in JavaScript-based web servers. In *USENIX Security Symposium*, pages 361–376. USENIX Association, 2018.

[37] Stack overflow outage postmortem, 2016. Accessed: 2023-08-30. URL: https://stackstatus.net/post/147710624694/outage-postmortem-july-20-2016.

[38] Cloudflare's outage postmortem, 2019. Accessed: 2023-08-30. URL: https://blog.cloudflare.com/details-of-the-cloudflare-outage-on-july-2-2019/.

[39] National vulnerability database: CVE-2020-3899, 2020. Accessed: 2023-08-30. URL: https://nvd.nist.gov/vuln/detail/CVE-2020-3899.

[40] Francesco Parolini and Antoine Miné. rat - ReDoS Abstract Tester, 2022. URL: https://github.com/parof/rat.

[41] Hal Berghel. The code red worm. *Commun. ACM*, 44(12):15–19, 2001. doi:10.1145/501317.501328.

[42] Hilarie K. Orman. The morris worm: A fifteen-year perspective. *IEEE Secur. Priv.*, 1(5):35–43, 2003. doi:10.1109/MSECP.2003.1236233.

[43] E Schultz, Jim Mellander, and D Peterson. The MS-SQL slammer worm. *Network Security*, 2003(3):10–14, 2003. doi:https://doi.org/10.1016/S1353-4858(03)00310-6.

[44] Zakir Durumeric, James Kasten, David Adrian, J. Alex Halderman, Michael Bailey, Frank Li, Nicholas Weaver, Johanna Amann, Jethro Beekman, Mathias Payer, and Vern Paxson. The matter of heartbleed. In *Internet Measurement Conference, IMC*, pages 475–488. ACM, 2014. doi:10.1145/2663716.2663755.

[45] CVE-2022-36934. Available from NIST, CVE-ID CVE-2022-36934. Accessed: 2023-08-30. URL: https://nvd.nist.gov/vuln/detail/CVE-2022-36934.

[46] CVE-2019-8745. Available from NIST, CVE-ID CVE-2019-8745. Accessed: 2023-08-30. URL: https://nvd.nist.gov/vuln/detail/CVE-2019-8745.

[47] CVE-2022-4135. Available from NIST, CVE-ID CVE-2022-4135. Accessed: 2023-08-30. URL: https://nvd.nist.gov/vuln/detail/CVE-2022-4135.

[48] M. Journault, A. Miné, R. Monat, and A. Ouadjaout. Combinations of reusable abstract domains for a multilingual static analyzer. In *Proc. of the 11th Working Conference on Verified Software: Theories, Tools, and Experiments (VSTTE19)*, volume 12031 of *Lecture Notes in Computer Science (LNCS)*, pages 1–18. Springer, 2019. http://www-apr.lip6.fr/~mine/publi/article-mine-al-vstte19.pdf. doi:10.1007/978-3-030-41600-3_1.

[49] Juliet C/C++ test suite, 2017. Accessed: 2023-08-30. URL: https://samate.nist.gov/SARD/test-suites/112.

[50] Michael R. Clarkson and Fred B. Schneider. Hyperproperties. *21st IEEE Computer Security Foundations Symposium*, pages 51–65, 2008.

[51] Francesco Parolini and Antoine Miné. Sound static analysis of regular expressions for vulnerabilities to denial of service attacks. In *Theoretical Aspects of Software Engineering (TASE)*, pages 73–91. Springer International Publishing, 2022. doi:10.1007/978-3-031-10363-6_6.

[52] Francesco Parolini and Antoine Miné. Sound static analysis of regular expressions for vulnerabilities to denial of service attacks. *Science of Computer Programming*, 229:102960, 2023. doi:https://doi.org/10.1016/j.scico.2023.102960.

[53] Francesco Parolini and Antoine Miné. Sound abstract nonexploitability analysis. In *Verification, Model Checking, and Abstract Interpretation, VMCAI*, Lecture Notes in Computer Science. Springer, 2024. https://hal.science/hal-04268105.

[54] Francesco Parolini and Antoine Miné. Sound Abstract Nonexploitability Analysis Artifact, September 2023. doi:10.5281/zenodo.8334112.

[55] Francesco Parolini. Exploitability analysis in MOPSA, 2023. URL: https://gitlab.com/parof/mopsa-analyzer/-/tree/exploitability-c?ref_type=heads.

[56] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–310, 1955.

[57] Stephen Cole Kleene. Introduction to metamathematics. 1952.

[58] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to automata theory, languages, and computation, 3rd Edition*. Pearson international edition. Addison-Wesley, 2007.

[59] Michael Sipser. *Introduction to the Theory of Computation*. Course Technology, Boston, MA, third edition, 2013.

[60] Peter Linz. *An Introduction to Formal Languages and Automata (6th Edition)*. Jones & Bartlett Learning, 2016.

[61] Scott Owens, John Reppy, and Aaron Turon. Regular-expression derivatives re-examined. *Journal of Functional Programming*, 19(2):173–190, 2009. doi:10.1017/s0956796808007090.

[62] Jan Midtgaard, Flemming Nielson, and Hanne Riis Nielson. A parametric abstract domain for lattice-valued regular expressions. In *International Static Analysis Symposium, SAS*, volume 9837 of *Lecture Notes in Computer Science*, pages 338–360. Springer, 2016. doi:10.1007/978-3-662-53413-7_17.

[63] Janusz A. Brzozowski. Derivatives of regular expressions. *Journal of the ACM*, 11(4):481–494, 1964. doi:10.1145/321239.321249.

[64] Google's re2 library. Accessed: 2023-08-30. URL: https://github.com/google/re2.

[65] Russ Cox. Regular expression matching can be simple and fast, 2007.

[66] Victor Mikhaylovich Glushkov. The abstract theory of automata. *Russian Mathematical Surveys*, 16(5):1, 1961.

[67] Carl Chapman and Kathryn T. Stolee. Exploring regular expression usage and context in Python. In *International Symposium on Software Testing and Analysis, ISSTA*, pages 282–293. ACM, 2016. doi:10.1145/2931037.2931073.

[68] Louis G. Michael IV, James Donohue, James C. Davis, Dongyoon Lee, and Francisco Servant. Regexes are hard: Decision-making, difficulties, and risks in programming regular expressions. In *International Conference on Automated Software Engineering, ASE*, pages 415–426. IEEE, 2019. doi:10.1109/ASE.2019.00047.

[69] James Kirrage, Asiri Rathnayake, and Hayo Thielecke. Static analysis for regular expression denial-of-service attacks. In *International Conference of Network and System Security, NSS*, volume 7873 of *Lecture Notes in Computer Science*, pages 135–148. Springer, 2013. doi:10.1007/978-3-642-38631-2_11.

[70] Asiri Rathnayake and Hayo Thielecke. Static analysis for regular expression exponential runtime via substructural logics. *CoRR*, abs/1405.7058, 2014. arXiv:1405.7058.

[71] Nicolaas Weideman, Brink van der Merwe, Martin Berglund, and Bruce W. Watson. Analyzing matching time behavior of backtracking regular expression matchers by using ambiguity of NFA. In *International Conference on Implementation and Application of Automata, CIAA*, volume 9705 of *Lecture Notes in Computer Science*, pages 322–334. Springer, 2016. doi:10.1007/978-3-319-40946-7_27.

[72] Valentin Wüstholz, Oswaldo Olivo, Marijn J. H. Heule, and Isil Dillig. Static detection of dos vulnerabilities in programs that use regular expressions. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS*, volume 10206 of *Lecture Notes in Computer Science*, pages 3–20, 2017. doi:10.1007/978-3-662-54580-5_1.

[73] Yuju Shen, Yanyan Jiang, Chang Xu, Ping Yu, Xiaoxing Ma, and Jian Lu. ReScue: crafting regular expression DoS attacks. In *International Conference on Automated Software Engineering, ASE*, pages 225–235. ACM, 2018. doi:10.1145/3238147.3238159.

[74] The SonarSource tool. Accessed: 2023-08-30. URL: https://www.sonarsource.com/.

[75] The safe-regex tool. Accessed: 2023-08-30. URL: https://github.com/substack/safe-regex.

[76] The regexploit tool. Accessed: 2023-08-30. URL: https://github.com/doyensec/regexploit.

[77] The redos-detector tool. Accessed: 2023-08-30. URL: https://github.com/tjenkinson/redos-detector.

[78] Monica Lam, Ravi Sethi, JD Ullman, and Alfred Aho. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA, 2006.

[79] Jeffrey E. F. Friedl. *Mastering regular expressions - understand your data and be more productive: for Perl, PHP, Java, .NET, Ruby, and more (3. ed.)*. O'Reilly, 2006. URL: https://www.oreilly.com/library/view/mastering-regular-expressions/0596528124/.

[80] Félix López and Víctor Romero. *Mastering Python Regular Expressions*. Packt Publishing Ltd, 2014. URL: https://www.packtpub.com/product/mastering-python-regular-expressions/9781783283156.

[81] Alfred V. Aho. Algorithms for finding patterns in strings. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity*, pages 255–300. Elsevier and MIT Press, 1990.

[82] Akimasa Morihata. Translation of regular expression with lookahead into finite state automaton. *Computer Software*, 29(1):147–158, 2012. doi:10.11309/jssst.29.1_147.

[83] Takayuki Miyazaki and Yasuhiko Minamide. Derivatives of regular expressions with lookahead. *J. Inf. Process.*, 27:422–430, 2019. URL: https://doi.org/10.2197/ipsjjip.27.422, doi:10.2197/IPSJJIP.27.422.

[84] Konstantinos Mamouras and Agnishom Chattopadhyay. Efficient matching of regular expressions with lookaround assertions. *Proc. ACM Program. Lang.*, 8(POPL):2761–2791, 2024. doi:10.1145/3632934.

[85] Rust's regex matching engine. Accessed: 2023-03-08. URL: https://github.com/rust-lang/regex.

[86] Rust's regex module documentation. Accessed: 2023-03-08. URL: https://docs.rs/regex/latest/regex/.

[87] V8's regex matching engine. Accessed: 2023-03-08. URL: https://github.com/v8/v8/tree/11.3.116/src/regexp.

[88] V8 new matching engine announcement. Accessed: 2023-03-08. URL: https://blog.chromium.org/2009/02/irregexp-google-chromes-new-regexp.html.

[89] Martin Berglund, Frank Drewes, and Brink van der Merwe. Analyzing catastrophic backtracking behavior in practical regular expression matching. In *Automata and Formal Languages, AFL*, volume 151 of *EPTCS*, pages 109–123, 2014. doi:10.4204/EPTCS.151.7.

[90] Java's regex matching engine. Accessed: 2023-03-08. URL: https://github.com/openjdk/jdk/tree/jdk8-b120/jdk/src/share/classes/java/util/regex.

[91] Php's regex matching engine. Accessed: 2023-03-08. URL: https://github.com/php/php-src/tree/php-8.2.3/ext/pcre.

[92] PCRE2 regex engine documentation. Accessed: 2023-03-08. URL: https://www.pcre.org/current/doc/html/pcre2pattern.html.

[93] Perl's regex matching engine. Accessed: 2023-03-08. URL: https://github.com/Perl/perl5/blob/v5.37.9/regexec.c.

[94] Perl's regex module documentation. Accessed: 2023-03-08. URL: https://perldoc.perl.org/perlre.

[95] Python's regex matching engine. Accessed: 2023-03-08. URL: https://github.com/python/cpython/tree/3.11/Lib/re.

[96] Python's regex module documentation. Accessed: 2023-03-08. URL: https://docs.python.org/3/library/re.html.

[97] Ruby's regex matching engine. Accessed: 2023-03-08. URL: https://github.com/ruby/ruby/blob/v3_2_1/re.c.

[98] Ruby's regex module documentation. Accessed: 2023-03-08. URL: https://ruby-doc.org/core-2.7.0/Regexp.html.

[99] Valentin Antimirov. Partial derivatives of regular expressions and finite automaton constructions. *Theoretical Computer Science*, 155(2):291–319, 1996. doi:https://doi.org/10.1016/0304-3975(95)00182-4.

[100] The Vim text editor. Accessed: 2023-03-08. URL: https://github.com/vim/vim.

[101] POSIX Standard for Regular Expressions. Accessed: 2023-08-30. URL: https://pubs.opengroup.org/onlinepubs/009696899/basedefs/xbd_chap09.html.

[102] Regexlib database. Accessed: 2023-08-30. URL: https://regexlib.com/.

[103] Theofilos Petsios, Jason Zhao, Angelos D. Keromytis, and Suman Jana. Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities. In *Conference on Computer and Communications Security, CCS*, pages 2155–2168. ACM, 2017. doi:10.1145/3133956.3134073.

[104] Node package manager. Accessed: 2023-08-30. URL: https://www.npmjs.com/.

[105] Martin Berglund and Brink van der Merwe. On the semantics of regular expression parsing in the wild. *Theoretical Computer Science*, 679:69–82, 2017. doi:10.1016/j.tcs.2016.09.006.

[106] Cyril Allauzen, Mehryar Mohri, and Ashish Rastogi. General algorithms for testing the ambiguity of finite automata and the double-tape ambiguity of finite-state transducers. *International Journal of Foundations of Computer Science*, 22(04):883–904, June 2011. doi:10.1142/s0129054111008477.

[107] Andreas Weber and Helmut Seidl. On the degree of ambiguity of finite automata. *Theorertical Computer Science*, 88(2):325–349, 1991. doi:10.1016/0304-3975(91)90381-B.

[108] Brendan Cody-Kenny, Michael Fenton, Adrian Ronayne, Eoghan Considine, Thomas McGuire, and Michael O'Neill. A search for improved performance in regular expressions. In *Genetic and Evolutionary Computation Conference, GECCO*, pages 1280–1287, 2017. doi:10.1145/3071178.3071196.

[109] Yeting Li, Zhiwu Xu, Jialun Cao, Haiming Chen, Tingjian Ge, Shing-Chi Cheung, and Haoren Zhao. Flashregex: Deducing anti-redos regexes from examples. In *International Conference on Automated Software Engineering, ASE 2020*, pages 659–671, 2020. doi:10.1145/3324884.3416556.

[110] Rong Pan, Qinheping Hu, Gaowei Xu, and Loris D'Antoni. Automatic repair of regular expressions. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):139:1–139:29, 2019. doi:10.1145/3360565.

[111] J. C. Davis, F. Servant, and D. Lee. Using selective memoization to defeat regular expression denial of service (ReDoS). In *IEEE Symposium on Security and Privacy, SP*, pages 543–559. IEEE Computer Society, 2021. doi:10.1109/SP40001.2021.00032.

[112] Cheng-Hung Lin, Chen-Hsiung Liu, and Shih-Chieh Chang. Accelerating regular expression matching using hierarchical parallel machines on

GPU. In *Global Communications Conference, GLOBECOM*, pages 1–5, 2011. doi:10.1109/GLOCOM.2011.6133663.

[113] Xiaodong Yu and Michela Becchi. GPU acceleration of regular expression matching for large datasets: exploring the implementation space. In *Computing Frontiers Conference, CF*, pages 18:1–18:10, 2013. doi:10.1145/2482767.2482791.

[114] Michela Becchi and Srihari Cadambi. Memory-efficient regular expression search using state merging. In *Joint Conference of the IEEE Computer and Communications Societies, INFOCOM*, pages 1064–1072, 2007. doi:10.1109/INFCOM.2007.128.

[115] The snort database. http://www.snort.org/, 2020. Accessed: 2023-08-30.

[116] The pypi packet manager. https://pypi.org/. Accessed: 2023-08-30.

[117] James C. Davis, Christy A. Coghlan, Francisco Servant, and Dongyoon Lee. The impact of regular expression denial of service (ReDoS) in practice: an empirical study at the ecosystem scale. In *Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE*, pages 246–256. ACM, 2018. doi:10.1145/3236024.3236027.

[118] C language standard. Accessed: 2023-10-05. URL: https://www.iso.org/standard/29237.html.

[119] ECMASCRIPT 2023 language specification. Accessed: 2023-10-05. URL: https://www.ecma-international.org/publications-and-standards/standards/ecma-262/.

[120] Hanne Riis Nielson and Flemming Nielson. *Semantics with applications*, volume 104. Springer, 1992.

[121] Patrick Cousot. Constructive design of a hierarchy of semantics of a transition system by abstract interpretation. *Theor. Comput. Sci.*, 277(1-2):47–103, 2002. doi:10.1016/S0304-3975(00)00313-3.

[122] Bowen Alpern and Fred B. Schneider. Recognizing safety and liveness. *Distributed Comput.*, 2(3):117–126, 1987. doi:10.1007/BF01782772.

[123] Ellis S. Cohen. Information transmission in computational systems. In *Symposium on Operating System Principles, SOSP*, pages 133–139. ACM, 1977. doi:10.1145/800214.806556.

[124] Joseph A. Goguen and José Meseguer. Security policies and security models. In *Security and Privacy*, pages 11–20. IEEE Computer Society, 1982. doi:10.1109/SP.1982.10014.

[125] Joseph A. Goguen and José Meseguer. Unwinding and inference control. In *Security and Privacy*, pages 75–87. IEEE Computer Society, 1984. doi:10.1109/SP.1984.10019.

[126] Dorothy E. Denning and Peter J. Denning. Certification of programs for secure information flow. *Commun. ACM*, 20(7):504–513, 1977. doi:10.1145/359636.359712.

[127] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick D. McDaniel. Flowdroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Programming Language Design and Implementation, PLDI*, pages 259–269. ACM, 2014. doi:10.1145/2594291.2594299.

[128] Alan Mathison Turing et al. On computable numbers, with an application to the entscheidungsproblem. *J. of Math*, 58(345-363):5, 1936.

[129] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer, 2004. doi:10.1007/978-3-662-07964-5.

[130] Leonardo de Moura and Sebastian Ullrich. The lean 4 theorem prover and programming language. In *Automated Deduction, CADE*, volume 12699 of *Lecture Notes in Computer Science*, pages 625–635. Springer, 2021. doi:10.1007/978-3-030-79876-5_37.

[131] Makarius Wenzel, Lawrence C. Paulson, and Tobias Nipkow. The isabelle framework. In *Theorem Proving in Higher Order Logics, TPHOLs*, volume 5170 of *Lecture Notes in Computer Science*, pages 33–38. Springer, 2008. doi:10.1007/978-3-540-71067-7_7.

[132] Ana Bove, Peter Dybjer, and Ulf Norell. A brief overview of agda - a functional language with dependent types. In *Theorem Proving in Higher Order*

Logics, TPHOLs, volume 5674 of *Lecture Notes in Computer Science*, pages 73–78. Springer, 2009. doi:10.1007/978-3-642-03359-9_6.

[133] Nikhil Swamy, Catalin Hritcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean Karim Zinzindohoue, and Santiago Zanella Béguelin. Dependent types and multi-monadic effects in F. In *Symposium on Principles of Programming Languages, POPL*, pages 256–270. ACM, 2016. doi:10.1145/2837614.2837655.

[134] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning, LPAR*, volume 6355 of *Lecture Notes in Computer Science*, pages 348–370. Springer, 2010. doi:10.1007/978-3-642-17511-4_20.

[135] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 - where programs meet provers. In *Programming Languages and Systems, ESOP*, volume 7792 of *Lecture Notes in Computer Science*, pages 125–128. Springer, 2013. doi:10.1007/978-3-642-37036-6_8.

[136] Bernard Carré and Johnathan Randall Garnsworthy. SPARK - an annotated ada subset for safety-critical programming. In *TRI-ADA*, pages 392–402. ACM, 1990. doi:10.1145/255471.255563.

[137] Patrick Baudin, François Bobot, David Bühler, Loïc Correnson, Florent Kirchner, Nikolai Kosmatov, André Maroneze, Valentin Perrelle, Virgile Prevosto, Julien Signoles, and Nicky Williams. The dogged pursuit of bug-free C programs: the frama-c software analysis platform. *Commun. ACM*, 64(8):56–68, 2021. doi:10.1145/3470569.

[138] Fish Wang and Yan Shoshitaishvili. Angr - the next generation of binary analysis. In *IEEE Cybersecurity Development, SecDev*, pages 8–9. IEEE Computer Society, 2017. doi:10.1109/SECDEV.2017.14.

[139] Galois Inc. Crucible. Accessed: 2023-11-09. URL: https://github.com/GaloisInc/crucible.

[140] Robin David, Sébastien Bardin, Thanh Dinh Ta, Laurent Mounier, Josselin Feist, Marie-Laure Potet, and Jean-Yves Marion. BINSEC/SE: a dynamic symbolic execution toolkit for binary-level analysis. In *Software Analysis, Evolution, and Reengineering, SANER*, pages 653–656. IEEE Computer Society, 2016. doi:10.1109/SANER.2016.43.

[141] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Operating Systems Design and Implementation, OSDI*, pages 209–224. USENIX Association, 2008. URL: http://www.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf.

[142] Kenneth L. McMillan. *Symbolic model checking*. Kluwer, 1993. doi:10.1007/978-1-4615-3190-6.

[143] Dirk Beyer and M. Erkan Keremoglu. CPAchecker: A tool for configurable software verification. In *Computer Aided Verification, CAV*, volume 6806 of *Lecture Notes in Computer Science*, pages 184–190. Springer, 2011. doi:10.1007/978-3-642-22110-1_16.

[144] Matthias Heizmann, Yu-Fang Chen, Daniel Dietsch, Marius Greitschus, Jochen Hoenicke, Yong Li, Alexander Nutz, Betim Musa, Christian Schilling, Tanja Schindler, and Andreas Podelski. Ultimate automizer and the search for perfect interpolants - (competition contribution). In *Tools and Algorithms for the Construction and Analysis of Systems, TACAS*, volume 10806 of *Lecture Notes in Computer Science*, pages 447–451. Springer, 2018. doi:10.1007/978-3-319-89963-3_30.

[145] Daniel Kroening, Peter Schrammel, and Michael Tautschnig. CBMC: the C bounded model checker. *CoRR*, abs/2302.02384, 2023. arXiv:2302.02384, doi:10.48550/ARXIV.2302.02384.

[146] Antoine Miné. Tutorial on static inference of numeric invariants by abstract interpretation. *Foundations and Trends in Programming Languages (FnTPL)*, 4(3–4):120–372, 2017. http://www-apr.lip6.fr/~mine/publi/article-mine-FTiPL17.pdf. doi:10.1561/2500000034.

[147] Xavier Rival and Kwangkeun Yi. *Introduction to static analysis: an abstract interpretation perspective*. Mit Press, 2020.

[148] Caterina Urban. The abstract domain of segmented ranking functions. In *Static Analysis Symposium, SAS*, volume 7935 of *Lecture Notes in Computer Science*, pages 43–62. Springer, 2013. doi:10.1007/978-3-642-38856-9_5.

[149] Caterina Urban and Antoine Miné. An abstract domain to infer ordinal-valued ranking functions. In Zhong Shao, editor, *Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014, Held as Part of*

the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings, volume 8410 of *Lecture Notes in Computer Science*, pages 412–431. Springer, 2014. doi:10.1007/978-3-642-54833-8_22.

[150] Caterina Urban. *Static Analysis by Abstract Interpretation of Functional Temporal Properties of Programs. (Analyse Statique par Interprétation Abstraite de Propriétés Temporelles Fonctionnelles des Programmes)*. PhD thesis, École Normale Supérieure, Paris, France, 2015. URL: https://tel.archives-ouvertes.fr/tel-01176641.

[151] Frédéric Benhamou, Frédéric Goualard, Laurent Granvilliers, and Jean-Francois Puget. Revising hull and box consistency. In *International Conference on Logic Programming, ICLP*, pages 230–244. MIT Press, 1999.

[152] R. Monat, M. Milanese, F. Parolini, J. Boillot, A. Ouadjaout, and A. Miné. Mopsa-c: Improved verification for c programs, simple validation of correctness witnesses (competition contribution). 2024.

[153] Arie Gurfinkel and Sagar Chaki. Boxes: A symbolic abstract domain of boxes. In *Static Analysis Symposium, SAS*, volume 6337 of *Lecture Notes in Computer Science*, pages 287–303. Springer, 2010. doi:10.1007/978-3-642-15769-1_18.

[154] Philippe Granger. Static analysis of arithmetical congruences. *International Journal of Computer Mathematics*, 30(3-4):165–190, 1989.

[155] Philippe Granger. Static analyses of congruence properties on rational numbers (extended abstract). In *Static Analysis Symposium, SAS*, volume 1302 of *Lecture Notes in Computer Science*, pages 278–292. Springer, 1997. doi:10.1007/BFB0032748.

[156] Michael Karr. Affine relationships among variables of a program. *Acta Informatica*, 6:133–151, 1976. doi:10.1007/BF00268497.

[157] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Principles of Programming Languages, POPL*, pages 84–96. ACM Press, 1978. doi:10.1145/512760.512770.

[158] Antoine Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation (HOSC)*, 19(1):31–100, 2006. http://www-apr.lip6.fr/~mine/publi/article-mine-HOSC06.pdf. doi:10.1007/s10990-006-8609-1.

[159] A. Miné. Field-sensitive value analysis of embedded C programs with union types and pointer arithmetics. In *Proc. of the ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'06)*, pages 54–63. ACM, 2006. http://www-apr.lip6.fr/~mine/publi/article-mine-lctes06.pdf.

[160] A. Miné. Static analysis by abstract interpretation of concurrent programs. Technical report, École normale supérieure, May 2013. http://www-apr.lip6.fr/~mine/hdr/hdr-compact-col.pdf.

[161] Gogul Balakrishnan and Thomas W. Reps. Recency-abstraction for heap-allocated storage. In *Static Analysis Symposium, SAS*, volume 4134 of *Lecture Notes in Computer Science*, pages 221–239. Springer, 2006. doi:10.1007/11823230_15.

[162] Raphaël Monat. *Static type and value analysis by abstract interpretation of Python programs with native C libraries. (Analyse statique, de type et de valeur, par interprétation abstraite, de programmes Python utilisant des librairies C)*. PhD thesis, Sorbonne University, Paris, France, 2021. URL: https://tel.archives-ouvertes.fr/tel-03533030.

[163] Denis Gopan, Thomas W. Reps, and Shmuel Sagiv. A framework for numeric analysis of array operations. In Jens Palsberg and Martín Abadi, editors, *Principles of Programming Languages, POPL*, pages 338–350. ACM, 2005. doi:10.1145/1040305.1040333.

[164] Sumit Gulwani, Bill McCloskey, and Ashish Tiwari. Lifting abstract interpreters to quantified logical domains. In George C. Necula and Philip Wadler, editors, *Principles of Programming Languages, POPL*, pages 235–246. ACM, 2008. doi:10.1145/1328438.1328468.

[165] Nicolas Halbwachs and Mathias Péron. Discovering properties about arrays in simple programs. In Rajiv Gupta and Saman P. Amarasinghe, editors, *Conference on Programming Language Design and Implementation, PLDI*, pages 339–348. ACM, 2008. doi:10.1145/1375581.1375623.

[166] Patrick Cousot, Radhia Cousot, and Francesco Logozzo. A parametric segmentation functor for fully automatic and scalable array content analysis. In Thomas Ball and Mooly Sagiv, editors, *Principles of Programming Languages, POPL*, pages 105–118. ACM, 2011. doi:10.1145/1926385.1926399.

[167] Raphaël Monat, Abdelraouf Ouadjaout, and Antoine Miné. Static type analysis by abstract interpretation of python programs. In *European Conference on Object-Oriented Programming, ECOOP*, volume 166 of *LIPIcs*, pages 17:1–17:29. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPICS.ECOOP.2020.17.

[168] Simon Holm Jensen, Anders Møller, and Peter Thiemann. Type analysis for javascript. In *Static Analysis Symposium, SAS*, volume 5673 of *Lecture Notes in Computer Science*, pages 238–255. Springer, 2009. doi:10.1007/978-3-642-03237-0_17.

[169] Francesco Logozzo. *Analyse statique modulaire des langages à objet. (Modular static analysis of object-oriented languages)*. PhD thesis, École Polytechnique, Palaiseau, France, 2004. URL: https://tel.archives-ouvertes.fr/pastel-00000896.

[170] Bor-Yuh Evan Chang and Xavier Rival. Relational inductive shape analysis. In George C. Necula and Philip Wadler, editors, *Principles of Programming Languages, POPL*, pages 247–260. ACM, 2008. doi:10.1145/1328438.1328469.

[171] Ahmed Bouajjani, Cezara Dragoi, Constantin Enea, and Mihaela Sighireanu. Abstract domains for automated reasoning about list-manipulating programs with infinite data. In Viktor Kuncak and Andrey Rybalchenko, editors, *Verification, Model Checking, and Abstract Interpretation, VMCAI*, volume 7148 of *Lecture Notes in Computer Science*, pages 1–22. Springer, 2012. doi:10.1007/978-3-642-27940-9_1.

[172] Josselin Giet, Félix Ridoux, and Xavier Rival. A product of shape and sequence abstractions. In Manuel V. Hermenegildo and José F. Morales, editors, *Static Analysis Symposium, SAS*, volume 14284 of *Lecture Notes in Computer Science*, pages 310–342. Springer, 2023. doi:10.1007/978-3-031-44245-2_15.

[173] Matthieu Journault, Antoine Miné, and Abdelraouf Ouadjaout. An abstract domain for trees with numeric relations. In Luís Caires, editor, *European Symposium on Programming, ESOP*, volume 11423 of *Lecture Notes in Computer Science*, pages 724–751. Springer, 2019. doi:10.1007/978-3-030-17184-1_26.

[174] Nurit Dor, Michael Rodeh, and Shmuel Sagiv. CSSV: towards a realistic tool for statically detecting all buffer overflows in C. In Ron Cytron and Rajiv

Gupta, editors, *Programming Language Design and Implementation, PLDI*, pages 155–167. ACM, 2003. doi:10.1145/781131.781149.

[175] Tae-Hyoung Choi, Oukseh Lee, Hyunha Kim, and Kyung-Goo Doh. A practical string analyzer by the widening approach. In Naoki Kobayashi, editor, *Asian Symposium of Programming Languages and Systems, APLAS*, volume 4279 of *Lecture Notes in Computer Science*, pages 374–388. Springer, 2006. doi:10.1007/11924661_23.

[176] Magnus Madsen and Esben Andreasen. String analysis for dynamic field access. In *Compiler Construction, CC*, volume 8409 of *Lecture Notes in Computer Science*, pages 197–217. Springer, 2014. doi:10.1007/978-3-642-54807-9_12.

[177] Changhee Park, Hyeonseung Im, and Sukyoung Ryu. Precise and scalable static analysis of jQuery using a regular expression domain. In *Proceedings of the 12th Symposium on Dynamic Languages, DLS 2016, Amsterdam, The Netherlands, November 1, 2016*, pages 25–36. ACM, 2016. doi:10.1145/2989225.2989228.

[178] Vincenzo Arceri, Isabella Mastroeni, and Sunyi Xu. Static analysis for ECMAScript string manipulation programs. *Applied Sciences*, 10(10):3525, 2020.

[179] Vincenzo Arceri, Mila Dalla Preda, Roberto Giacobazzi, and Isabella Mastroeni. SEA: string executability analysis by abstract interpretation. *CoRR*, abs/1702.02406, 2017. arXiv:1702.02406.

[180] Luca Negrini, Vincenzo Arceri, Pietro Ferrara, and Agostino Cortesi. Twinning automata and regular expressions for string static analysis. In *Verification, Model Checking, and Abstract Interpretation - 22nd International Conference, VMCAI 2021, Copenhagen, Denmark, January 17-19, 2021, Proceedings*, volume 12597 of *Lecture Notes in Computer Science*, pages 267–290. Springer, 2021. doi:10.1007/978-3-030-67067-2_13.

[181] Vincenzo Arceri, Martina Olliaro, Agostino Cortesi, and Pietro Ferrara. Relational string abstract domains. In Bernd Finkbeiner and Thomas Wies, editors, *Verification, Model Checking, and Abstract Interpretation, VMCAI*, volume 13182 of *Lecture Notes in Computer Science*, pages 20–42. Springer, 2022. doi:10.1007/978-3-030-94583-1_2.

[182] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software. In *The Essence of Computation, Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones [on occasion of his 60th birthday]*, volume 2566 of *Lecture Notes in Computer Science*, pages 85–108. Springer, 2002. doi:10.1007/3-540-36377-7_5.

[183] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The Astrée analyzer. In *European Symposium on Programming ESOP*, volume 3444 of *Lecture Notes in Computer Science (LNCS)*, pages 21–30. Springer, 2005. http://www-apr.lip6.fr/~mine/publi/esop05_astree.pdf. doi:10.1007/978-3-540-31987-0_3.

[184] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, and Xavier Rival. Why does astrée scale up? *Formal Methods Syst. Des.*, 35(3):229–264, 2009. doi:10.1007/S10703-009-0089-6.

[185] Manuel Fähndrich and Francesco Logozzo. Static contract checking with abstract interpretation. In *Formal Verification of Object-Oriented Software, FoVeOOS*, volume 6528 of *Lecture Notes in Computer Science*, pages 10–30. Springer, 2010. doi:10.1007/978-3-642-18070-5_2.

[186] David Bühler. *Structuring an Abstract Interpreter through Value and State Abstractions:EVA, an Evolved Value Analysis for Frama-C. (Structurer un interpréteur abstrait au moyen d'abstractions de valeurs et d'états :Eva, une analyse de valeur évoluée pour Frama-C)*. PhD thesis, University of Rennes 1, France, 2017. URL: https://tel.archives-ouvertes.fr/tel-01664726.

[187] Sandrine Blazy, David Bühler, and Boris Yakobowski. Structuring abstract interpreters through state and value abstractions. In *Verification, Model Checking, and Abstract Interpretation, VMCAI*, volume 10145 of *Lecture Notes in Computer Science*, pages 112–130. Springer, 2017. doi:10.1007/978-3-319-52234-0_7.

[188] Guillaume Brat, Jorge A. Navas, Nija Shi, and Arnaud Venet. IKOS: a framework for static analysis based on abstract interpretation. In *Software Engineering and Formal Methods, SEFM*, volume 8702 of *Lecture Notes in Computer Science*, pages 271–277. Springer, 2014. doi:10.1007/978-3-319-10431-7_20.

[189] Chris Lattner and Vikram S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, CGO*, pages 75–88. IEEE Computer Society, 2004. doi:10.1109/CGO.2004.1281665.

[190] Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A. Navas. The seahorn verification framework. In *Computer Aided Verification, CAV*, volume 9206 of *Lecture Notes in Computer Science*, pages 343–361. Springer, 2015. doi:10.1007/978-3-319-21690-4_20.

[191] Dino Distefano, Manuel Fähndrich, Francesco Logozzo, and Peter W. O'Hearn. Scaling static analyses at facebook. *Commun. ACM*, 62(8):62–70, 2019. doi:10.1145/3338112.

[192] The Infer team. Infer: A static analyzer for Java, C, C++, and Objective-C, 2019. Accessed: 2023-12-04. URL: https://github.com/facebook/infer.

[193] Fausto Spoto. The julia static analyzer for java. In *Static Analysis Symposium, SAS*, volume 9837 of *Lecture Notes in Computer Science*, pages 39–57. Springer, 2016. doi:10.1007/978-3-662-53413-7_3.

[194] Fausto Spoto, Elisa Burato, Michael D. Ernst, Pietro Ferrara, Alberto Lovato, Damiano Macedonio, and Ciprian Spiridon. Static identification of injection attacks in java. *ACM Trans. Program. Lang. Syst.*, 41(3):18:1–18:58, 2019. doi:10.1145/3332371.

[195] Pietro Ferrara, Luca Negrini, Vincenzo Arceri, and Agostino Cortesi. Static analysis for dummies: experiencing lisa. In *State Of the Art in Program Analysis, SOAP*, pages 1–6. ACM, 2021. doi:10.1145/3460946.3464316.

[196] Luca Negrini, Pietro Ferrara, Vincenzo Arceri, and Agostino Cortesi. LiSA: a generic framework for multilanguage static analysis. In *Challenges of Software Verification*, pages 19–42. Springer, 2023.

[197] Luca Olivieri, Fabio Tagliaferro, Vincenzo Arceri, Marco Ruaro, Luca Negrini, Agostino Cortesi, Pietro Ferrara, Fausto Spoto, and Enrico Talin. Ensuring determinism in blockchain software with GoLiSA: an industrial experience report. In *State Of the Art in Program Analysis, SOAP*, pages 23–29. ACM, 2022. doi:10.1145/3520313.3534658.

[198] Abdelraouf Ouadjaout and Antoine Miné. A library modeling language for the static analysis of C programs. In *Static Analysis Symposium, SAS*, volume

12389 of *Lecture Notes in Computer Science*, pages 223–247. Springer, 2020. doi:10.1007/978-3-030-65474-0_11.

[199] Raphaël Monat, Abdelraouf Ouadjaout, and Antoine Miné. A multilanguage static analysis of python programs with native C extensions. In *Static Analysis Symposium, SAS*, volume 12913 of *Lecture Notes in Computer Science*, pages 323–345. Springer, 2021. doi:10.1007/978-3-030-88806-0_16.

[200] B. Jeannet and A. Miné. Apron: A library of numerical abstract domains for static analysis. In *Computer Aided Verification, CAV*, volume 5643 of *Lecture Notes in Computer Science (LNCS)*, pages 661–667. Springer, 2009. http://www-apr.lip6.fr/~mine/publi/article-mine-jeannet-cav09.pdf. doi:10.1007/978-3-642-02658-4_52.

[201] The Pysa static analyzer. URL: https://engineering.fb.com/2020/08/07/security/pysa/.

[202] Li Li, Tegawendé F. Bissyandé, Mike Papadakis, Siegfried Rasthofer, Alexandre Bartel, Damien Octeau, Jacques Klein, and Yves Le Traon. Static analysis of android apps: a systematic literature review. *Inf. Softw. Technol.*, 88:67–95, 2017. doi:10.1016/j.infsof.2017.04.001.

[203] Simon Holm Jensen, Anders Møller, and Peter Thiemann. Interprocedural analysis with lazy propagation. In *Static Analysis Symposium, SAS*, volume 6337 of *Lecture Notes in Computer Science*, pages 320–339. Springer, 2010. doi:10.1007/978-3-642-15769-1_20.

[204] Simon Holm Jensen, Magnus Madsen, and Anders Møller. Modeling the HTML DOM and browser API in static analysis of JavaScript web applications. In *Symposium on the Foundations of Software Engineering, SIGSOFT*, pages 59–69. ACM, 2011. doi:10.1145/2025113.2025125.

[205] Simon Holm Jensen, Peter A. Jonsson, and Anders Møller. Remedying the eval that men do. In *International Symposium on Software Testing and Analysis, ISSTA*, pages 34–44. ACM, 2012. doi:10.1145/2338965.2336758.

[206] Erik Krogh Kristensen and Anders Møller. Reasonably-most-general clients for javascript library analysis. In *International Conference on Software Engineering, ICSE*, pages 83–93. IEEE / ACM, 2019. doi:10.1109/ICSE.2019.00026.

[207] Jacques-Henri Jourdan, Vincent Laporte, Sandrine Blazy, Xavier Leroy, and David Pichardie. A formally-verified C static analyzer. In *Symposium on*

Principles of Programming Languages, POPL, pages 247–259. ACM, 2015. doi:10.1145/2676726.2676966.

[208] Sylvie Boldo, Jacques-Henri Jourdan, Xavier Leroy, and Guillaume Melquiond. A formally-verified C compiler supporting floating-point arithmetic. In *Symposium on Computer Arithmetic, ARITH*, pages 107–115. IEEE Computer Society, 2013. doi:10.1109/ARITH.2013.30.

[209] Jacques-Henri Jourdan. *Verasco: a Formally Verified C Static Analyzer. (Verasco: un analyseur statique pour C formellement vérifié)*. PhD thesis, Paris Diderot University, France, 2016. URL: https://tel.archives-ouvertes.fr/tel-01327023.

[210] Sandrine Blazy, Vincent Laporte, and David Pichardie. Verified abstract interpretation techniques for disassembling low-level self-modifying code. *J. Autom. Reason.*, 56(3):283–308, 2016. doi:10.1007/S10817-015-9359-8.

[211] Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, 2009. doi:10.1145/1538788.1538814.

[212] 2023 CWE top 25 most dangerous software weaknesses, 2023. Accessed: 2023-08-30. URL: https://cwe.mitre.org/top25/archive/2023/2023_top25_list.html.

[213] Guillaume Girol, Benjamin Farinier, and Sébastien Bardin. Not all bugs are created equal, but robust reachability can tell the difference. In *Computer Aided Verification, CAV*, volume 12759, pages 669–693. Springer, 2021. doi:10.1007/978-3-030-81685-8_32.

[214] The Infer static analyzer. URL: https://fbinfer.com/.

[215] Isabella Mastroeni and Michele Pasqua. Hyperhierarchy of semantics - A formal framework for hyperproperties verification. In *Static Analysis Symposium, SAS*, volume 10422, pages 232–252, 2017. doi:10.1007/978-3-319-66706-5_12.

[216] Isabella Mastroeni and Michele Pasqua. Verifying bounded subset-closed hyperproperties. In *Static Analysis Symposium, SAS*, volume 11002, pages 263–283, 2018. doi:10.1007/978-3-319-99725-4_17.

[217] Isabella Mastroeni and Michele Pasqua. Statically analyzing information flows: an abstract interpretation-based hyperanalysis for non-interference. In *Symposium on Applied Computing, SAC*, pages 2215–2223, 2019. doi:10.1145/3297280.3297498.

[218] Caterina Urban and Peter Müller. An abstract interpretation framework for input data usage. In *European Symposium on Programming, ESOP*, volume 10801, pages 683–710, 2018. doi:10.1007/978-3-319-89884-1_24.

[219] Mounir Assaf, David A. Naumann, Julien Signoles, Eric Totel, and Frédéric Tronel. Hypercollecting semantics and its application to static analysis of information flow. In *Principles of Programming Languages, POPL*, 2017. doi:10.1145/3009837.3009889.

[220] Peter Ørbæk and Jens Palsberg. Trust in the lambda-calculus. *J. Funct. Program.*, 7(6):557–591, 1997. doi:10.1017/s0956796897002906.

[221] Andrew C. Myers and Barbara Liskov. A decentralized model for information flow control. In *Symposium on Operating System Principles, SOSP*, pages 129–142. ACM, 1997. doi:10.1145/268998.266669.

[222] Dennis M. Volpano, Cynthia E. Irvine, and Geoffrey Smith. A sound type system for secure flow analysis. *Journal of Computer Security*, 4(2/3):167–188, 1996. doi:10.3233/JCS-1996-42-304.

[223] Nevin Heintze and Jon G. Riecke. The slam calculus: Programming with secrecy and integrity. In *Principles of Programming Languages, POPL*, pages 365–377. ACM, 1998. doi:10.1145/268946.268976.

[224] Geoffrey Smith and Dennis M. Volpano. Secure information flow in a multi-threaded imperative language. In *Principles of Programming Languages, POPL*, pages 355–364. ACM, 1998. doi:10.1145/268946.268975.

[225] Dennis M. Volpano and Geoffrey Smith. Probabilistic noninterference in a concurrent language. *J. Comput. Secur.*, 7(1), 1999. doi:10.3233/jcs-1999-72-305.

[226] Johan Agat. Transforming out timing leaks. In *Principles of Programming Languages, POPL*, pages 40–53. ACM, 2000. doi:10.1145/325694.325702.

[227] Andrei Sabelfeld and David Sands. Probabilistic noninterference for multi-threaded programs. In *Computer Security Foundations Workshop, CSFW*, pages 200–214. IEEE Computer Society, 2000. doi:10.1109/CSFW.2000.856937.

[228] Steve Zdancewic and Andrew C. Myers. Secure information flow and CPS. In *European Symposium on Programming, ESOP*, volume 2028 of *Lecture Notes in Computer Science*, pages 46–61. Springer, 2001. doi:10.1007/3-540-45309-1_4.

[229] Anindya Banerjee and David A. Naumann. Secure information flow and pointer confinement in a java-like language. In *Computer Security Foundations Workshop CSFW*, page 253. IEEE Computer Society, 2002. doi:10.1109/CSFW.2002.1021820.

[230] François Pottier and Vincent Simonet. Information flow inference for ML. *ACM Trans. Program. Lang. Syst.*, 25(1):117–158, 2003. doi:10.1145/596980.596983.

[231] Torben Amtoft and Anindya Banerjee. Information flow analysis in logical form. In Roberto Giacobazzi, editor, *Static Analysis Symposium, SAS*, volume 3148 of *Lecture Notes in Computer Science*, pages 100–115. Springer, 2004. doi:10.1007/978-3-540-27864-1_10.

[232] Roberto Giacobazzi and Isabella Mastroeni. Abstract non-interference: parameterizing non-interference by abstract interpretation. In Neil D. Jones and Xavier Leroy, editors, *Principles of Programming Languages, POPL*, pages 186–197. ACM, 2004. doi:10.1145/964001.964017.

[233] Isabella Mastroeni. On the rôle of abstract non-interference in language-based security. In Kwangkeun Yi, editor, *Programming Languages and Systems, Third Asian Symposium, APLAS 2005, Tsukuba, Japan, November 2-5, 2005, Proceedings*, volume 3780 of *Lecture Notes in Computer Science*, pages 418–433. Springer, 2005. doi:10.1007/11575467_27.

[234] Roberto Giacobazzi and Isabella Mastroeni. Generalized abstract non-interference: Abstract secure information-flow analysis for automata. In Vladimir Gorodetsky, Igor V. Kotenko, and Victor A. Skormin, editors, *Workshop on Mathematical Methods, Models, and Architectures for Computer Network Security, MMM-ACNS*, volume 3685 of *Lecture Notes in Computer Science*, pages 221–234. Springer, 2005. doi:10.1007/11560326_17.

[235] Roberto Giacobazzi and Isabella Mastroeni. Adjoining classified and unclassified information by abstract interpretation. *J. Comput. Secur.*, 18(5):751–797, 2010. doi:10.3233/JCS-2009-0382.

[236] Roberto Giacobazzi and Isabella Mastroeni. A proof system for abstract non-interference. *J. Log. Comput.*, 20(2):449–479, 2010. doi:10.1093/LOGCOM/EXP053.

[237] Isabella Mastroeni and Anindya Banerjee. Modelling declassification policies using abstract domain completeness. *Math. Struct. Comput. Sci.*, 21(6):1253–1299, 2011. doi:10.1017/S096012951100020X.

[238] Roberto Giacobazzi and Isabella Mastroeni. Abstract non-interference: A unifying framework for weakening information-flow. *ACM Trans. Priv. Secur.*, 21(2):9:1–9:31, 2018. doi:10.1145/3175660.

[239] Marc Éluard and Thomas P. Jensen. Secure object flow analysis for Java Card. In Peter Honeyman, editor, *Smart Card Research and Advanced Application Conference, CARDIS*, pages 97–110. USENIX, 2002. URL: http://www.usenix.org/publications/library/proceedings/cardis02/eluard.html.

[240] Zhiqun Chen. *Java Card technology for smart cards: architecture and programmer's guide*. Addison-Wesley Professional, 2000.

[241] David Cachera, Thomas P. Jensen, David Pichardie, and Vlad Rusu. Extracting a data flow analyser in constructive logic. In David A. Schmidt, editor, *European Symposium on Programming, ESOP*, volume 2986 of *Lecture Notes in Computer Science*, pages 385–400. Springer, 2004. doi:10.1007/978-3-540-24725-8_27.

[242] Frédéric Besson, Thomas P. Jensen, and Pierre Vittet. SawjaCard: A static analysis tool for certifying Java Card applications. In Markus Müller-Olm and Helmut Seidl, editors, *Static Analysis Symposium, SAS*, volume 8723 of *Lecture Notes in Computer Science*, pages 51–67. Springer, 2014. doi:10.1007/978-3-319-10936-7_4.

[243] Gilles Barthe, Pedro R. D'Argenio, and Tamara Rezk. Secure information flow by self-composition. *Mathematical Structures in Computer Science*, 21(6):1207–1252, 2011. doi:10.1017/S0960129511000193.

[244] Tachio Terauchi and Alexander Aiken. Secure information flow as a safety problem. In *Static Analysis Symposium, SAS*, volume 3672 of *Lecture Notes in Computer Science*, pages 352–367, 2005. doi:10.1007/11547662_24.

[245] Timos Antonopoulos, Paul Gazzillo, Michael Hicks, Eric Koskinen, Tachio Terauchi, and Shiyi Wei. Decomposition instead of self-composition for proving the absence of timing channels. In *Conference on Programming Language Design and Implementation, PLDI*, pages 362–375. ACM, 2017. doi:10.1145/3062341.3062378.

[246] Bernd Finkbeiner, Markus N. Rabe, and César Sánchez. Algorithms for model checking hyperltl and hyperctl ^*. In *Computer Aided Verification, CAV*, volume 9206 of *Lecture Notes in Computer Science*, pages 30–48. Springer, 2015. doi:10.1007/978-3-319-21690-4_3.

[247] Michael R. Clarkson, Bernd Finkbeiner, Masoud Koleini, Kristopher K. Micinski, Markus N. Rabe, and César Sánchez. Temporal logics for hyperproperties. In *Principles of Security and Trust, POST*, volume 8414 of *Lecture Notes in Computer Science*, pages 265–284. Springer, 2014. doi:10.1007/978-3-642-54792-8_15.

[248] Norine Coenen, Bernd Finkbeiner, César Sánchez, and Leander Tentrup. Verifying hyperliveness. In *Computer Aided Verification, CAV*, volume 11561 of *Lecture Notes in Computer Science*, pages 121–139. Springer, 2019. doi:10.1007/978-3-030-25540-4_7.

[249] Bernd Finkbeiner, Christopher Hahn, and Hazem Torfah. Model checking quantitative hyperproperties. In *Computer Aided Verification, CAV*, volume 10981 of *Lecture Notes in Computer Science*, pages 144–163. Springer, 2018. doi:10.1007/978-3-319-96145-3_8.

[250] Tzu-Han Hsu, César Sánchez, and Borzoo Bonakdarpour. Bounded model checking for hyperproperties. In *Tools and Algorithms for the Construction and Analysis of Systems, TACAS*, volume 12651 of *Lecture Notes in Computer Science*, pages 94–112. Springer, 2021. doi:10.1007/978-3-030-72016-2_6.

[251] Raven Beutner and Bernd Finkbeiner. Autohyper: Explicit-state model checking for hyperltl. In *Tools and Algorithms for the Construction and Analysis of Systems, TACAS*, volume 13993 of *Lecture Notes in Computer Science*, pages 145–163. Springer, 2023. doi:10.1007/978-3-031-30823-9_8.

[252] Patrick Cousot. Abstract semantic dependency. In *Static Analysis Symposium, SAS*, volume 11822, pages 389–410. Springer, 2019. doi:10.1007/978-3-030-32304-2_19.

[253] Ignacio Tiraboschi, Tamara Rezk, and Xavier Rival. Sound symbolic execution via abstract interpretation and its application to security. In *Verification, Model Checking, and Abstract Interpretation, VMCAI*, volume 13881 of *Lecture Notes in Computer Science*, pages 267–295. Springer, 2023. doi:10.1007/978-3-031-24950-1_13.

[254] Chaoqiang Deng and Patrick Cousot. Responsibility analysis by abstract interpretation. In Bor-Yuh Evan Chang, editor, *Static Analysis Symposium, SAS*, volume 11822 of *Lecture Notes in Computer Science*, pages 368–388. Springer, 2019. doi:10.1007/978-3-030-32304-2_18.

[255] Chaoqiang Deng. *Responsibility Analysis by Abstract Interpretation*. PhD thesis, New York University, USA, 2021. URL: https://cs.nyu.edu/media/publications/NYU_dissertation_Deng.pdf.

[256] Chaoqiang Deng and Patrick Cousot. The systematic design of responsibility analysis by abstract interpretation. *ACM Trans. Program. Lang. Syst.*, 44(1):3:1–3:90, 2022. doi:10.1145/3484938.

[257] David Bühler, Pascal Cuoq, Boris Yakobowski, Matthieu Lemerre, André Maroneze, Valentin Perrelle, and Virgile Prevosto. *The Eva plug-in*. CEA List, November 2023. Version 28.0. URL: https://frama-c.com/download/frama-c-eva-manual.pdf.

[258] Mark D. Weiser. Program slicing. *IEEE Trans. Software Eng.*, 10(4):352–357, 1984. doi:10.1109/TSE.1984.5010248.

[259] Frank Tip. A survey of program slicing techniques. *J. Program. Lang.*, 3(3), 1995. URL: http://compscinet.dcs.kcl.ac.uk/JP/jp030301.abs.html.

[260] David W. Binkley and Keith Brian Gallagher. Program slicing. *Adv. Comput.*, 43:1–50, 1996. doi:10.1016/S0065-2458(08)60641-5.

[261] Andrea De Lucia. Program slicing: Methods and applications. In *Workshop on Source Code Analysis and Manipulation (SCAM)*, pages 144–151. IEEE Computer Society, 2001. doi:10.1109/SCAM.2001.972675.

[262] Keith Brian Gallagher and James R. Lyle. Using program slicing in software maintenance. *IEEE Trans. Software Eng.*, 17(8):751–761, 1991. doi:10.1109/32.83912.

[263] Gerardo Canfora, Aniello Cimitile, and Andrea De Lucia. Conditioned program slicing. *Inf. Softw. Technol.*, 40(11-12):595–607, 1998. doi:10.1016/S0950-5849(98)00086-X.

[264] John Field, G. Ramalingam, and Frank Tip. Parametric program slicing. In Ron K. Cytron and Peter Lee, editors, *Principles of Programming Languages, POPL*, pages 379–392. ACM Press, 1995. doi:10.1145/199448.199534.

[265] Aniello Cimitile, Andrea De Lucia, and Malcolm Munro. A specification driven slicing process for identifying reusable functions. *J. Softw. Maintenance Res. Pract.*, 8(3):145–178, 1996.

[266] Susan Horwitz, Thomas W. Reps, and David W. Binkley. Interprocedural slicing using dependence graphs. *ACM Trans. Program. Lang. Syst.*, 12(1):26–60, 1990. doi:10.1145/77606.77608.

[267] Xavier Rival. Understanding the origin of alarms in astrée. In Chris Hankin and Igor Siveroni, editors, *Static Analysis Symposium, SAS*, volume 3672 of *Lecture Notes in Computer Science*, pages 303–319. Springer, 2005. doi:10.1007/11547662_21.

[268] Torben Amtoft and Anindya Banerjee. A logic for information flow analysis with an application to forward slicing of simple imperative programs. *Sci. Comput. Program.*, 64(1):3–28, 2007. doi:10.1016/J.SCICO.2006.03.002.

[269] Hyoung Seok Hong, Insup Lee, and Oleg Sokolsky. Abstract slicing: A new approach to program slicing based on abstract interpretation and model checking. In *Workshop on Source Code Analysis and Manipulation, (SCAM)*, pages 25–34. IEEE Computer Society, 2005. doi:10.1109/SCAM.2005.2.

[270] Isabella Mastroeni and Damiano Zanardini. Abstract program slicing: An abstract interpretation-based approach to program slicing. *ACM Trans. Comput. Log.*, 18(1):7:1–7:58, 2017. doi:10.1145/3029052.

[271] Isabella Mastroeni and Damiano Zanardini. Data dependencies and program slicing: from syntax to abstract semantics. In Robert Glück and Oege de Moor, editors, *Partial Evaluation and Semantics-based Program Manipulation, PEPM*, pages 125–134. ACM, 2008. doi:10.1145/1328408.1328428.

[272] Damiano Zanardini. The semantics of abstract program slicing. In *Workshop on Source Code Analysis and Manipulation (SCAM)*, pages 89–98. IEEE Computer Society, 2008. doi:10.1109/SCAM.2008.19.

[273] Isabella Mastroeni and Durica Nikolic. Abstract program slicing: From theory towards an implementation. In Jin Song Dong and Huibiao Zhu, editors, *International Conference on Formal Engineering Methods, ICFEM*, volume 6447 of *Lecture Notes in Computer Science*, pages 452–467. Springer, 2010. doi:10.1007/978-3-642-16901-4_30.

[274] Yanis Sellami, Guillaume Girol, Frédéric Recoules, Damien Couroussé, and Sébastien Bardin. Inference of robust reachability constraints. 8(POPL):2731–2760, 2024. doi:10.1145/3632933.

[275] Adnan Aziz, Kumud Sanwal, Vigyan Singhal, and Robert K. Brayton. Verifying continuous time markov chains. In *Computer Aided Verification, CAV*, volume 1102 of *Lecture Notes in Computer Science*, pages 269–276. Springer, 1996. doi:10.1007/3-540-61474-5_75.

[276] Hans Hansson and Bengt Jonsson. A logic for reasoning about time and reliability. *Formal Aspects Comput.*, 6(5):512–535, 1994. doi:10.1007/BF01211866.

[277] Patrick Cousot and Michael Monerau. Probabilistic abstract interpretation. In *European Symposium on Programming, ESOP*, volume 7211 of *Lecture Notes in Computer Science*, pages 169–193. Springer, 2012. doi:10.1007/978-3-642-28869-2_9.

[278] David Monniaux. Abstract interpretation of probabilistic semantics. In *Static Analysis Symposium, SAS*, volume 1824 of *Lecture Notes in Computer Science*, pages 322–339. Springer, 2000. doi:10.1007/978-3-540-45099-3_17.

[279] David Monniaux. An abstract analysis of the probabilistic termination of programs. In *Static Analysis Symposium, SAS*, volume 2126 of *Lecture Notes in Computer Science*, pages 111–126. Springer, 2001. doi:10.1007/3-540-47764-0_7.

[280] Alessandra Di Pierro and Herbert Wiklicky. Probabilistic abstract interpretation: From trace semantics to dtmc's and linear regression. In *Semantics, Logics, and Calculi - Essays Dedicated to Hanne Riis Nielson and Flemming Nielson on the Occasion of Their 60th Birthdays*, volume 9560 of *Lecture Notes in Computer Science*, pages 111–139. Springer, 2016. doi:10.1007/978-3-319-27810-0_6.

[281] Sébastien Bardin and Guillaume Girol. A quantitative flavour of robust reachability. *CoRR*, abs/2212.05244, 2022. arXiv:2212.05244, doi:10.48550/arXiv.2212.05244.

[282] Jonathan Heusser and Pasquale Malacaria. Quantifying information leaks in software. In *Annual Computer Security Applications Conference, ACSAC*, pages 261–269. ACM, 2010. doi:10.1145/1920261.1920300.

[283] A. Ouadjaout and A. Miné. A library modeling language for the static analysis of C programs. In *Static Analysis Symposium, SAS*, volume 12389 of *Lecture*

Notes in Computer Science (LNCS), pages 223–246. Springer, 2020. http://www-apr.lip6.fr/~mine/publi/ouadjaout-al-sas20.pdf. doi:10.1007/978-3-030-65474-0_11.

[284] Common weakness enumeration (CWE). Accessed: 2023-08-30. URL: https://cwe.mitre.org/.

Appendix

Appendix A

Proofs

Proofs for Chapter 4

We define the *frontier* of a state $(\mathcal{R}, w)$ as the (possibly empty) ordered sequence of states that are reached after matching the first character of the word w. More formally, the frontier $\texttt{front} : \mathbb{S}_r \to \mathbb{S}_r^*$ is the sequence of states

$$\texttt{front}((\mathcal{R}, aw)) \triangleq (\texttt{refresh}(\mathcal{R}_1), w), \ldots, (\texttt{refresh}(\mathcal{R}_n), w)$$

Where $[\![a\mathcal{R}_1]\!](aw), \ldots, [\![a\mathcal{R}_n]\!](aw)$ is the (possibly empty) ordered sequence of subtrees of $[\![\mathcal{R}]\!](aw)$ such that the next action is matching the first character a. For example, $\texttt{front}(((a \mid a)^*, ab)) = ((a \mid a)^*, b), ((a \mid a)^*, b)$ (see Figure 4.2B). We define $\texttt{front}((\mathcal{R}, \epsilon))$ as the empty sequence. We abuse the notation and we generalize the frontier to sequences of states: $\texttt{front}((\mathcal{R}_1, w), \ldots, (\mathcal{R}_n, w))$ is the ordered concatenation of the frontiers $\texttt{front}((\mathcal{R}_1, w)), \ldots, \texttt{front}((\mathcal{R}_n, w))$.

Proof (Lemma 4.1)
Let m be the least integer such that $\texttt{front}^m((\mathcal{R}, w))$ is empty. Since the number of nodes between the frontiers does not depend on the length of the input word but only on the regular expression, $h = \Theta(m)$. Let $n \leq m$, and observe that the words in the states of $\texttt{front}^n((\mathcal{R}, w))$ have exactly $|w| - n$ characters. By definition of $\texttt{front}$, when $n = |w|$ the next frontier must be empty. This implies that m cannot be greater than $|w|$, so that $h = \mathcal{O}(|w|)$.

We now prove the correctness of Algorithm 2, namely we show $\mathcal{L}(\texttt{M2}(\mathcal{R})) = \mathcal{M}_2(\mathcal{R})$ (see Thm. 4.1). Since `M2` immediately calls `M2-rec`, we actually prove that $\mathcal{L}(\texttt{M2-rec}(\mathcal{R}, \emptyset)) = \mathcal{M}_2(\mathcal{R})$. First, we introduce some preliminary definitions, and then we formalize the correctness theorem for `M2-rec`. Then, we proceed to prove by induction that `M2-rec` is correct. Finally, we observe that the correctness of `M2` is a corollary of the correctness of `M2-rec`.

Before proving the correctness of `M2`, we need some preliminary definitions. We define the set of *reachable regular expressions* $\texttt{rch} : \mathbb{R}^{\mathcal{T}} \to \wp(\mathbb{R}^{\mathcal{T}})$ as follows.

$$\texttt{rch}(\mathcal{R}) \triangleq \{\, \mathcal{R}' \in \mathbb{R}^{\mathcal{T}} \mid \exists w_1, w_2 \in \Sigma^*, \exists t \in \mathcal{T}((\mathcal{R}, w_1 w_2)) : \ell(t) = (\mathcal{R}', w_2) \,\}$$

Let $\mathcal{R}_1, \mathcal{R}_2 \in \mathbb{R}^{\mathcal{T}}$, $w_1, w_2 \in \Sigma^*$. If $\exists t \in \mathcal{T}((\mathcal{R}_1, w_1 w_2))$ such that $(\mathcal{R}_2, w_2) = \ell(t)$, then we write $(\mathcal{R}_1, w_1 w_2) \longrightarrow^* (\mathcal{R}_2, w_2)$. We need to define when a regular expression $\mathcal{R} \in \mathbb{R}^{\mathcal{T}}$ is *valid*, namely when it is possible to obtain it by following a series of transitions from an initial expression in $\mathbb{R}$. We say that $\mathcal{R} \in \mathbb{R}^{\mathcal{T}}$ is *valid* iff $\exists \mathcal{R}_1 \in \mathbb{R}, w_1, w_2 \in \Sigma^*$ such that $(\mathcal{R}_1, w_1 w_2) \longrightarrow^* (\mathcal{R}, w_2)$. Consider as example $ab^{\overline{*}}$: there is no regex in $\mathbb{R}$ that can produce a concatenated with $b^{\overline{*}}$, so that $ab^{\overline{*}}$ is not valid.

Let S be a nonempty set of regular expressions such that $\forall \mathcal{R}_1, \mathcal{R}_2 \in S$ if $\mathcal{R}_1 \neq \mathcal{R}_2$, then $|\mathcal{R}_1| \neq |\mathcal{R}_2|$ (where $|\mathcal{R}|$ is the number of constructors in the expression). We extract the longest element of S with the function $\mathrm{L} : \wp(\mathbb{R}^{\mathcal{T}}) \to \mathbb{R}^{\mathcal{T}}$ defined as $\mathrm{L}(S) \triangleq \operatorname{argmax}_{\mathcal{R} \in S} |\mathcal{R}|$. Let $\mathcal{R}' \in \mathbb{R}^{\mathcal{T}}, \mathcal{R} = \mathcal{R}_1 \cdots \mathcal{R}_n \in \mathbb{R}^{\mathcal{T}}$ where for all $i \in [1 \ldots n]$, $\mathcal{R}_i$ is not a concatenation. We define a function to determine whether a regular expression is a suffix of another modulo $^{\overline{*}}$. $\mathrm{suff} : \mathbb{R}^{\mathcal{T}} \times \mathbb{R}^{\mathcal{T}} \to \mathbb{B}$ is defined as $\mathrm{suff}(\mathcal{R}', \mathcal{R}_1 \cdots \mathcal{R}_n) = \texttt{tt}$ iff $\exists j \in [1 \ldots n]$ such that $\texttt{refresh}(\mathcal{R}') = \texttt{refresh}(\mathcal{R}_{j+1} \ldots \mathcal{R}_n)$. For example, $\mathrm{suff}(a^* a, a a^{\overline{*}} a) = \texttt{tt}$. We say that a set S is a *valid set of expansion of nested stars* if: (1) $\forall \mathcal{R} \in S, \exists \mathcal{R}_1, \mathcal{R}_2 \in \mathbb{R}^{\mathcal{T}}$ such that $\mathcal{R} = \mathcal{R}_1^* \mathcal{R}_2$; (2) $\forall \mathcal{R}_1, \mathcal{R}_2 \in S$ such that $\mathcal{R}_1 \neq \mathcal{R}_2$ it holds $|\mathcal{R}_1| \neq |\mathcal{R}_2|$; (3) $\forall \mathcal{R} \in S \setminus \{\mathrm{L}(S)\} : \mathrm{suff}(\mathcal{R}, \mathrm{L}(S))$. An example of a valid set of expansion of nested stars is $\{\, (a^*)^*, a^*(a^*)^{\overline{*}} \,\}$.

Let $\mathcal{R}_1, \mathcal{R}_2 \in \mathbb{R}^{\mathcal{T}}$. If $\mathcal{R}_1 = \mathcal{R}_2$, then we define $\mathcal{M}_2^{\mathcal{R}_2} : \mathbb{R} \to \Sigma^*$ as $\mathcal{M}_2^{\mathcal{R}_2}(\mathcal{R}_1) \triangleq \emptyset$. If $\mathcal{R}_1 \neq \mathcal{R}_2$, $\mathcal{M}_2^{\mathcal{R}_2}(\mathcal{R}_1)$ is defined as follows.

$$\mathcal{M}_2^{\mathcal{R}_2}(\mathcal{R}_1) \triangleq \{\, w_1 w_2 \mid w_1 \in \Sigma^+, w_2 \in \Sigma^*, \exists t_1, t_2 \in \mathcal{T}((\mathcal{R}_1, w_1 w_2)) : t_1 \neq t_2 \wedge \ell(t_1) = \ell(t_2) = (\mathcal{R}_2, w_2) \wedge w_2 \in \mathcal{L}(\mathcal{R}_2) \,\}$$

The words in $\mathcal{M}_2^{\mathcal{R}_2}(\mathcal{R}_1)$ are those that can reach $\mathcal{R}_2$ in at least two different traces from $\mathcal{R}_1$, and then can be matched from $\mathcal{R}_2$.

We now formalize the correctness of `M2-rec`. We define a precondition for `M2-rec`, and then we give a postcondition. The correctness theorem, namely Thm. A.1, states that the precondition implies the postcondition. Let $\mathcal{R} \in \mathbb{R}^{\mathcal{T}}$, $E \in \wp(\mathbb{R}^{\mathcal{T}})$.

Precondition

1. $\mathcal{R}$ is valid
2. E is a valid set of expansion of nested stars
3. $\forall \mathcal{R}_i \in E$ it holds that $\text{suff}(\mathcal{R}_i, \mathcal{R})$

The precondition asserts that the actual arguments of the calls to `M2-rec` are consistent: it forbids calling the function with an arbitrary set of regular expressions as argument E. In particular, the second and the third conditions together ensure that E is obtained by expanding the stars in $\mathcal{R}$. This is always verified if `M2-rec` is initially invoked with E set to $\emptyset$. Observe that the precondition trivially holds for $\texttt{M2-rec}(\mathcal{R}, \emptyset)$ if $\mathcal{R} \in \mathbb{R}$. Furthermore, if $\mathcal{R} \in E$, then $\mathcal{R} = \text{L}(E)$.

Postcondition

- If $E = \emptyset$, then $\mathcal{L}(\texttt{M2-rec}(\mathcal{R}, E)) = \mathcal{M}_2(\mathcal{R})$;
- If $E \neq \emptyset$, then $\mathcal{M}_2^{\text{L}(E)}(\mathcal{R}) \subseteq \mathcal{L}(\texttt{M2-rec}(\mathcal{R}, E)) \subseteq \mathcal{M}_2(\mathcal{R})$.

The first case in the postcondition specifies that if E is empty, then the language recognized by $\texttt{M2-rec}(\mathcal{R}, E)$ is exactly $\mathcal{M}_2(\mathcal{R})$. The second condition is more interesting, as it corresponds to the case in which E is not empty, namely the algorithm is expanding the body of a star. In this case, the function returns an overapproximation of the words that have a nonempty prefix that is matched in at least two different traces and can then reach $\text{L}(E)$, which is the star that the algorithm is expanding.

Theorem A.1 (Correctness of `M2-rec`)
If the precondition holds for `M2-rec`, then the postcondition holds.

Thm. A.1 formalizes that if `M2-rec` is called with correct parameters, then it computes $\mathcal{M}_2$. In case the algorithm is expanding a star (that is, $E \neq \emptyset$), it computes the language of words that have a nonempty prefix that is matched in at least two different traces and can then reach the star.

Observe that if $\mathcal{R} \in \mathbb{R}$, the precondition holds for `M2-rec`$(\mathcal{R}, \emptyset)$. This implies that we can apply the correctness theorem and obtain that (by the second case in the postcondition) $\mathcal{L}(\texttt{M2-rec}(\mathcal{R}, \emptyset)) = \mathcal{M}_2(\mathcal{R})$. As mentioned at the beginning of this Section, this is equivalent to $\mathcal{L}(\texttt{M2}(\mathcal{R})) = \mathcal{M}_2(\mathcal{R})$, which is the statement of Thm. 4.1. In Corollary A.1 we formalize that the correctness of `M2` is a corollary of Thm. A.1.

We prove that the precondition implies the postcondition by induction on the set of actual arguments that will be used in the subcalls of `M2-rec`. First, we formally define this set. Given the call `M2-rec`$(\mathcal{R}, E)$, we can associate to it the set of pairs $A(\mathcal{R}, E)$ such that for each $(\mathcal{R}_1, E_1) \in A(\mathcal{R}, E)$ it holds (1) `M2-rec`$(\mathcal{R}_1, E_1)$ is called in a subcall of `M2-rec`$(\mathcal{R}, E)$; (2) the control flow reaches line 6 (that is, $\mathcal{R}_1$ has not been expanded yet). $A(\mathcal{R}, E)$ is the set of actual arguments that will be used in the subcalls of `M2-rec`$(\mathcal{R}, E)$. It can be proved that for each $\mathcal{R} \in \mathbb{R}^{\mathcal{T}}, E \in \wp(\mathbb{R}^{\mathcal{T}})$ that respect the precondition, it holds that $(\mathcal{R}, E) \notin A(\mathcal{R}, E)$, namely the configuration $(\mathcal{R}, E)$ will never be expanded again in any subcall of `M2-rec`$(\mathcal{R}, E)$. This is because the algorithm keeps track, with the formal parameter E, of stars that have already been analyzed, and as soon as a regular expression that has a star as the first construct in the concatenation is encountered for the second time, the function terminates at line 5, never reaching line 6.

Furthermore, we observe that $A(\mathcal{R}, E)$ is a finite set. This is because for each $(\mathcal{R}_1, E_1) \in A(\mathcal{R}, E)$ it holds that $\mathcal{R}_1 \in \texttt{rch}(\mathcal{R})$ (since the algorithm explores all regular expressions that can be expanded during the concrete execution), and `rch`$(\mathcal{R})$ is finite. The finiteness of $A(\mathcal{R}, E)$ and the fact that $(\mathcal{R}, E) \notin A(\mathcal{R}, E)$ imply the termination of the algorithm and show that the induction is well-founded.

Proof (Thm. A.1)
We prove by induction on $A(\mathcal{R}, E)$ that the precondition always implies the postcondition.

Base Case ($A(\mathcal{R}, E) = \emptyset$)

If $A(\mathcal{R}, E) = \emptyset$, then there are no subcalls to `M2-rec`. There are only three possible cases.

1. $(\texttt{regex-head}(\mathcal{R}), \texttt{regex-tail}(\mathcal{R})) = (\epsilon, \epsilon)$. Then, the execution reaches line 8 and $\perp_r$ is correctly returned, since no word in Σ^+ is matched in two different traces from ϵ.

2. $(\texttt{regex-head}(\mathcal{R}), \texttt{regex-tail}(\mathcal{R})) = (\mathcal{R}_1^{\overline{*}}, \mathcal{R}_2)$. Then, similarly to the previous case, the execution reaches line 8 and $\perp_r$ is returned, since no word can be matched if the first constructor in the concatenation is $^{\overline{*}}$.

3. $\mathcal{R} \in E$. Then, by the second and third conditions in the precondition it must be that $\mathcal{R} = \mathrm{L}(E)$. By definition, $\mathcal{M}_2^{\mathcal{R}}(\mathcal{R}) = \emptyset$, and we conclude by observing that we correctly return $\perp_r$ at line 5.

Inductive Case ($A(\mathcal{R}, E) \neq \emptyset$)**,** $E = \emptyset$

If $A(\mathcal{R}, E) \neq \emptyset$, then we are in the inductive case and there are subcalls to `M2-rec`. We first consider the subcase in which $E = \emptyset$, that is the algorithm is not expanding any star. We show that $\mathcal{L}(\texttt{M2-rec}(\mathcal{R}, E)) = \mathcal{M}_2(\mathcal{R})$ in three different cases that depend on $\mathcal{R}$.

1. $(\texttt{regex-head}(\mathcal{R}), \texttt{regex-tail}(\mathcal{R})) = (a, \mathcal{R}_1)$. In this case we return $a \cdot \texttt{M2-rec}(\texttt{refresh}(\mathcal{R}_1), \emptyset)$. Since the precondition is satisfied for $\texttt{M2-rec}(a\mathcal{R}_1, \emptyset)$, it is satisfied also for $\texttt{M2-rec}(\texttt{refresh}(\mathcal{R}_1), \emptyset)$. Furthermore, since $(\texttt{refresh}(\mathcal{R}_1), \emptyset) \notin A(\texttt{refresh}(\mathcal{R}_1), \emptyset)$, we have $A(\texttt{refresh}(\mathcal{R}_1), \emptyset) \subset A(a\mathcal{R}_1, \emptyset)$. We can then apply the inductive hypothesis:

$$\begin{aligned}
\mathcal{L}(a \cdot \texttt{M2-rec}(\texttt{refresh}(\mathcal{R}_1), \emptyset)) &= \mathcal{L}(a)\mathcal{M}_2(\texttt{refresh}(\mathcal{R}_1)) \\
&\qquad\qquad \text{(inductive hypothesis)} \\
&= \mathcal{M}_2(a \cdot \texttt{refresh}(\mathcal{R}_1)) \\
&= \mathcal{M}_2(a\mathcal{R}_1) \\
(\forall w \in \Sigma^* : \mathcal{T}((a \cdot \texttt{refresh}(\mathcal{R}_1), w)) &= \mathcal{T}((a\mathcal{R}_1, w)))
\end{aligned}$$

2. $(\texttt{regex-head}(\mathcal{R}), \texttt{regex-tail}(\mathcal{R})) = (\mathcal{R}_1 \mid \mathcal{R}_2, \mathcal{R}_3)$. The first action is a choice. We can divide $\mathcal{M}_2((\mathcal{R}_1 \mid \mathcal{R}_2)\mathcal{R}_3)$ in three subsets: (1) the words matched by both branches of the current choice, namely $\mathcal{L}(\mathcal{R}_1\mathcal{R}_3 \cap_{\not\epsilon} \mathcal{R}_2\mathcal{R}_3)$; (2) the words that are matched in at least two different traces after taking the left branch, namely $\mathcal{M}_2(\mathcal{R}_1\mathcal{R}_3)$; (3) the words that are matched in at least two different traces after taking the right branch, namely $\mathcal{M}_2(\mathcal{R}_2\mathcal{R}_3)$. Similarly to the previous case, the precondition in each subcall is satisfied. Furthermore, $A(\mathcal{R}_1\mathcal{R}_3, \emptyset) \subset A((\mathcal{R}_1 \mid \mathcal{R}_2)\mathcal{R}_3, \emptyset)$ and $A(\mathcal{R}_2\mathcal{R}_3, \emptyset) \subset A((\mathcal{R}_1 \mid \mathcal{R}_2)\mathcal{R}_3, \emptyset)$ hold. We can then apply the inductive hypothesis: $\mathcal{L}(\texttt{M2-rec}(\mathcal{R}_1\mathcal{R}_3, \emptyset))$ equals $\mathcal{M}_2(\mathcal{R}_1\mathcal{R}_3)$ and $\mathcal{L}(\texttt{M2-rec}(\mathcal{R}_2\mathcal{R}_3, \emptyset))$ equals $\mathcal{M}_2(\mathcal{R}_2\mathcal{R}_3)$. Observing that we return the regular expression $(\mathcal{R}_1\mathcal{R}_3 \cap_{\not\epsilon} \mathcal{R}_2\mathcal{R}_3) \mid \texttt{M2-rec}(\mathcal{R}_1\mathcal{R}_2, \emptyset) \mid \texttt{M2-rec}(\mathcal{R}_2\mathcal{R}_2, \emptyset)$, we can conclude:

$$
\begin{aligned}
&\mathcal{L}(\texttt{M2-rec}((\mathcal{R}_1 \mid \mathcal{R}_2)\mathcal{R}_3, \emptyset)) \\
&= \mathcal{L}((\mathcal{R}_1\mathcal{R}_3 \cap_{\not\epsilon} \mathcal{R}_2\mathcal{R}_3) \mid \texttt{M2-rec}(\mathcal{R}_1\mathcal{R}_3, \emptyset) \mid \texttt{M2-rec}(\mathcal{R}_2\mathcal{R}_3, \emptyset)) \\
&= \mathcal{L}(\mathcal{R}_1\mathcal{R}_3 \cap_{\not\epsilon} \mathcal{R}_2\mathcal{R}_3) \cup \mathcal{M}_2(\mathcal{R}_1\mathcal{R}_3) \cup \mathcal{M}_2(\mathcal{R}_2\mathcal{R}_3) \qquad \text{(inductive hypothesis)} \\
&= \mathcal{M}_2((\mathcal{R}_1 \mid \mathcal{R}_2)\mathcal{R}_3)
\end{aligned}
$$

3. $(\texttt{regex-head}(\mathcal{R}), \texttt{regex-tail}(\mathcal{R})) = (\mathcal{R}_1^*, \mathcal{R}_2)$. Then the first action is a choice. Similarly to the previous case, we can divide $\mathcal{M}_2(\mathcal{R}_1^*\mathcal{R}_2)$ in three subsets: (1) the words matched by both branches of the current choice (that is to expand the star or not), namely $\mathcal{L}(\mathcal{R}_1\overline{\mathcal{R}_1^*}\mathcal{R}_2 \cap_{\not\epsilon} \mathcal{R}_2)$; (2) the words that are matched in at least two different traces in the body of the star and that can reach $\mathcal{R}_1^*\mathcal{R}_2$, namely $\mathcal{M}_2^{\mathcal{R}_1^*\mathcal{R}_2}(\mathcal{R}_1\overline{\mathcal{R}_1^*}\mathcal{R}_2)$; (3) the words that are matched in at least two different traces in $\mathcal{R}_2$, namely $\mathcal{M}_2(\mathcal{R}_2)$. Observe that the words in $\mathcal{M}_2(\mathcal{R}_2)$ have as prefix language all the words that can be matched in $\mathcal{R}_1^*$, so that the last set actually is $\mathcal{L}(\mathcal{R}_1^*)\mathcal{M}_2(\mathcal{R}_2)$. If the precondition holds for $\texttt{M2-rec}(\mathcal{R}_1^*\mathcal{R}_2, \emptyset)$, then it holds for the subcalls. Furthermore, $A(\mathcal{R}_1\overline{\mathcal{R}_1^*}\mathcal{R}_2, \{\mathcal{R}_1^*\mathcal{R}_2\}) \subset A(\mathcal{R}_1^*\mathcal{R}_2, \emptyset)$ and $A(\mathcal{R}_2, \emptyset) \subset A(\mathcal{R}_1^*\mathcal{R}_2, \emptyset)$,

so that by inductive hypothesis we have:

$$\mathcal{L}(\mathcal{R}_1^* \cdot \texttt{M2-rec}(\mathcal{R}_2, \emptyset)) = \mathcal{L}(\mathcal{R}_1^*)\mathcal{M}_2(\mathcal{R}_2)$$

$$\mathcal{M}_2^{\mathcal{R}_1^*\mathcal{R}_2}(\mathcal{R}_1\mathcal{R}_1^*\mathcal{R}_2) \subseteq \mathcal{L}(\texttt{M2-rec}(\mathcal{R}_1\mathcal{R}_1^{\overline{*}}\mathcal{R}_2, \{\mathcal{R}_1^*\mathcal{R}_2\})) \subseteq \mathcal{M}_2(\mathcal{R}_1\mathcal{R}_1^{\overline{*}}\mathcal{R}_2)$$

Observing that we return $(\mathcal{R}_1\mathcal{R}_1^{\overline{*}}\mathcal{R}_2 \cap_{\not\epsilon} \mathcal{R}_2) \mid \texttt{M2-rec}(\mathcal{R}_1\mathcal{R}_1^{\overline{*}}\mathcal{R}_2, \{\mathcal{R}_1^*\mathcal{R}_2\}) \mid \mathcal{R}_1^* \cdot \texttt{M2-rec}(\mathcal{R}_2, \emptyset)$, we can conclude:

$$\begin{aligned}
&\mathcal{M}_2(\mathcal{R}_1^*\mathcal{R}_2) \\
&= \mathcal{L}(\mathcal{R}_1\mathcal{R}_1^{\overline{*}}\mathcal{R}_2 \cap_{\not\epsilon} \mathcal{R}_2) \cup \mathcal{M}_2^{\mathcal{R}_1^*\mathcal{R}_2}(\mathcal{R}_1\mathcal{R}_1^{\overline{*}}\mathcal{R}_2) \cup \mathcal{L}(\mathcal{R}_1^*)\mathcal{M}_2(\mathcal{R}_2) \\
&\subseteq \mathcal{L}((\mathcal{R}_1\mathcal{R}_1^{\overline{*}}\mathcal{R}_2 \cap_{\not\epsilon} \mathcal{R}_2) \\
&\qquad \mid \texttt{M2-rec}(\mathcal{R}_1\mathcal{R}_1^{\overline{*}}\mathcal{R}_2, \{\mathcal{R}_1^*\mathcal{R}_2\}) \mid \mathcal{R}_1^* \cdot \texttt{M2-rec}(\mathcal{R}_2, \emptyset)) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{(inductive hypothesis)} \\
&\subseteq \mathcal{L}(\mathcal{R}_1\mathcal{R}_1^{\overline{*}}\mathcal{R}_2 \cap_{\not\epsilon} \mathcal{R}_2) \cup \mathcal{M}_2(\mathcal{R}_1\mathcal{R}_1^{\overline{*}}\mathcal{R}_2) \cup \mathcal{L}(\mathcal{R}_1^*)\mathcal{M}_2(\mathcal{R}_2) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{(inductive hypothesis)} \\
&= \mathcal{M}_2(\mathcal{R}_1^*\mathcal{R}_2) \qquad\qquad (\mathcal{M}_2^{\mathcal{R}_1^*\mathcal{R}_2}(\mathcal{R}_1\mathcal{R}_1^*\mathcal{R}_2) \subseteq \mathcal{M}_2(\mathcal{R}_1\mathcal{R}_1^{\overline{*}}\mathcal{R}_2) \subseteq \mathcal{M}_2(\mathcal{R}_1^*\mathcal{R}_2))
\end{aligned}$$

So that $\mathcal{L}((\mathcal{R}_1\mathcal{R}_1^{\overline{*}}\mathcal{R}_2 \cap_{\not\epsilon} \mathcal{R}_2) \mid \texttt{M2-rec}(\mathcal{R}_1\mathcal{R}_1^{\overline{*}}\mathcal{R}_2, \{\mathcal{R}_1^*\mathcal{R}_2\}) \mid \mathcal{R}_1^* \cdot \texttt{M2-rec}(\mathcal{R}_2, \emptyset))$ equals $\mathcal{M}_2(\mathcal{R}_1^*\mathcal{R}_2)$.

Inductive Case $(A(\mathcal{R}, E) \neq \emptyset)$**,** $E \neq \emptyset$

We now consider the other subcase in the inductive case: $E \neq \emptyset$. In this case, the algorithm is expanding a star, and we show that $\mathcal{M}_2^{\mathrm{L}(E)}(\mathcal{R}) \subseteq \mathcal{L}(\texttt{M2-rec}(\mathcal{R}, E)) \subseteq \mathcal{M}_2(\mathcal{R})$. We prove this in three different cases that depend on $\mathcal{R}$.

1. $(\texttt{regex-head}(\mathcal{R}), \texttt{regex-tail}(\mathcal{R})) = (a, \mathcal{R}_1)$. In this case we return $a \cdot \texttt{M2-rec}(\texttt{refresh}(\mathcal{R}_1), E)$. By the fact that the precondition is satisfied for $\texttt{M2-rec}(\mathcal{R}, E)$, it is satisfied also for $\texttt{M2-rec}(\texttt{refresh}(\mathcal{R}_1), E)$. Furthermore, we have $A(\texttt{refresh}(\mathcal{R}_1), E) \subset A(a\mathcal{R}_1, E)$. We can then apply the inductive

hypothesis and obtain:

$$\begin{aligned}
\mathcal{M}_2^{\mathrm{L}(E)}(a\mathcal{R}_1) &= \mathcal{M}_2^{\mathrm{L}(E)}(a \cdot \texttt{refresh}(\mathcal{R}_1)) \\
&\qquad (\forall w \in \Sigma^* : \mathcal{T}((a\mathcal{R}_1, w)) = \mathcal{T}((a \cdot \texttt{refresh}(\mathcal{R}_1), w))) \\
&= \mathcal{L}(a)\mathcal{M}_2^{\mathrm{L}(E)}(\texttt{refresh}(\mathcal{R}_1)) \\
&\subseteq \mathcal{L}(a \cdot \texttt{M2-rec}(\texttt{refresh}(\mathcal{R}_1), E)) \qquad \text{(inductive hypothesis)} \\
&\subseteq \mathcal{L}(a)\mathcal{M}_2(\texttt{refresh}(\mathcal{R}_1)) \qquad \text{(inductive hypothesis)} \\
&= \mathcal{M}_2(a \cdot \texttt{refresh}(\mathcal{R}_1)) \\
&= \mathcal{M}_2(a\mathcal{R}_1) \\
&\qquad (\forall w \in \Sigma^* : \mathcal{T}((a \cdot \texttt{refresh}(\mathcal{R}_1), w)) = \mathcal{T}((a\mathcal{R}_1, w)))
\end{aligned}$$

2. $(\texttt{regex-head}(\mathcal{R}), \texttt{regex-tail}(\mathcal{R})) = (\mathcal{R}_1 \mid \mathcal{R}_2, \mathcal{R}_3)$. The first action is a choice. We can divide $\mathcal{M}_2^{\mathrm{L}(E)}((\mathcal{R}_1 \mid \mathcal{R}_2)\mathcal{R}_3)$ in three subsets: (1) the words in $\mathcal{M}_2^{\mathrm{L}(E)}(\mathcal{R}_1\mathcal{R}_3)$; (2) the words in $\mathcal{M}_2^{\mathrm{L}(E)}(\mathcal{R}_2\mathcal{R}_3)$; (3) the words w_1w_2 with $w_1 \in \Sigma^+$, $w_2 \in \Sigma^*$ such that $(\mathcal{R}_1\mathcal{R}_3, w_1w_2) \longrightarrow^* (\mathrm{L}(E), w_2)$, $(\mathcal{R}_2\mathcal{R}_3, w_1w_2) \longrightarrow^* (\mathrm{L}(E), w_2)$ and $w_2 \in \mathcal{L}(\mathrm{L}(E))$. This set corresponds to those words that have a nonempty prefix that can be matched by both branches of the alternative and can reach $\mathrm{L}(E)$. Let I be this set: observe that it is a subset of $\mathcal{L}(\mathcal{R}_1\mathcal{R}_3 \cap_{\notin} \mathcal{R}_2\mathcal{R}_3)$. The precondition in each subcall is satisfied, $A(\mathcal{R}_1\mathcal{R}_3, E) \subset A((\mathcal{R}_1 \mid \mathcal{R}_2)\mathcal{R}_3, E)$ and $A(\mathcal{R}_2\mathcal{R}_3, E) \subset A((\mathcal{R}_1 \mid \mathcal{R}_2)\mathcal{R}_3, E)$ hold. We can then apply the inductive hypothesis and, observing that we return $(\mathcal{R}_1\mathcal{R}_3 \cap_{\notin} \mathcal{R}_2\mathcal{R}_3) \mid \texttt{M2-rec}(\mathcal{R}_1\mathcal{R}_3, E) \mid \texttt{M2-rec}(\mathcal{R}_2\mathcal{R}_3, E)$, we obtain:

$$\begin{aligned}
&\mathcal{M}_2^{\mathrm{L}(E)}((\mathcal{R}_1 \mid \mathcal{R}_2)\mathcal{R}_3) \\
&= I \cup \mathcal{M}_2^{\mathrm{L}(E)}(\mathcal{R}_1\mathcal{R}_3) \cup \mathcal{M}_2^{\mathrm{L}(E)}(\mathcal{R}_2\mathcal{R}_3) \\
&\subseteq \mathcal{L}((\mathcal{R}_1\mathcal{R}_3 \cap_{\notin} \mathcal{R}_2\mathcal{R}_3) \mid \texttt{M2-rec}(\mathcal{R}_1\mathcal{R}_3, E) \mid \texttt{M2-rec}(\mathcal{R}_2\mathcal{R}_3, E)) \\
&\qquad \text{(inductive hypothesis and } I \subseteq \mathcal{L}(\mathcal{R}_1\mathcal{R}_3 \cap_{\notin} \mathcal{R}_2\mathcal{R}_3)) \\
&\subseteq \mathcal{L}((\mathcal{R}_1\mathcal{R}_3 \cap_{\notin} \mathcal{R}_2\mathcal{R}_3)) \cup \mathcal{M}_2(\mathcal{R}_1\mathcal{R}_3) \cup \mathcal{M}_2(\mathcal{R}_2\mathcal{R}_3) \\
&\qquad \text{(inductive hypothesis)} \\
&= \mathcal{M}_2((\mathcal{R}_1 \mid \mathcal{R}_2)\mathcal{R}_3) \qquad \text{(analogous to subcase } (\mathcal{R}_1 \mid \mathcal{R}_2)\mathcal{R}_3 \text{ if } E = \emptyset)
\end{aligned}$$

3. $(\texttt{regex-head}(\mathcal{R}), \texttt{regex-tail}(\mathcal{R})) = (\mathcal{R}_1^*, \mathcal{R}_2)$. The first action in this case is a choice. We can divide the set $\mathcal{M}_2^{\mathrm{L}(E)}(\mathcal{R}_1^*\mathcal{R}_2)$ in three subsets: (1) the words in $\mathcal{M}_2^{\mathrm{L}(E\cup\{\mathcal{R}_1^*\mathcal{R}_2\})}(\mathcal{R}_1\mathcal{R}_1^{\overline{*}}\mathcal{R}_2)$; (2) the words in $\mathcal{M}_2^{\mathrm{L}(E)}(\mathcal{R}_2)$ (they have as prefix language all the words that can be matched in $\mathcal{R}_1^*$, so that actually the set corresponds to $\mathcal{L}(\mathcal{R}_1^*)\mathcal{M}_2^{\mathrm{L}(E)}(\mathcal{R}_2)$); (3) the words w_1w_2 with $w_1 \in \Sigma^+, w_2 \in \Sigma^*$ such that we obtain $(\mathcal{R}_1\mathcal{R}_1^{\overline{*}}\mathcal{R}_2, w_1w_2) \longrightarrow^* (\mathrm{L}(E), w_2)$, $(\mathcal{R}_2, w_1w_2) \longrightarrow^* (\mathrm{L}(E), w_2)$ and $w_2 \in \mathcal{L}(\mathrm{L}(E))$. This set corresponds to those words that have a nonempty prefix that can be matched by both the expansion of the star and $\mathcal{R}_2$, and can then reach $\mathrm{L}(E)$. Let I be this set: observe that it is a subset of $\mathcal{L}(\mathcal{R}_1\mathcal{R}_1^{\overline{*}}\mathcal{R}_2 \cap_{\not\epsilon} \mathcal{R}_2)$. The precondition in each subcall is satisfied, $A(\mathcal{R}_1\mathcal{R}_1^*\mathcal{R}_2, E \cup \{\mathcal{R}_1^*\mathcal{R}_2\}) \subset A(\mathcal{R}_1^*\mathcal{R}_2, E)$ and $A(\mathcal{R}_2, E) \subset A(\mathcal{R}_1^*\mathcal{R}_2, E)$ hold. We can then apply the inductive hypothesis and, observing that we return $(\mathcal{R}_1\mathcal{R}_1^{\overline{*}}\mathcal{R}_2 \cap_{\not\epsilon} \mathcal{R}_2) \mid \texttt{M2-rec}(\mathcal{R}_1\mathcal{R}_1^{\overline{*}}\mathcal{R}_2, E \cup \{\mathcal{R}_1^*\mathcal{R}_2\}) \mid \mathcal{R}_1^* \cdot \texttt{M2-rec}(\mathcal{R}_2, E)$, we obtain:

$$
\begin{aligned}
&\mathcal{M}_2^{\mathrm{L}(E)}(\mathcal{R}_1^*\mathcal{R}_2) \\
&= I \cup \mathcal{M}_2^{\mathrm{L}(E\cup\{\mathcal{R}_1^*\mathcal{R}_2\})}(\mathcal{R}_1\mathcal{R}_1^{\overline{*}}\mathcal{R}_2) \cup \mathcal{L}(\mathcal{R}_1^*)\mathcal{M}_2^{\mathrm{L}(E)}(\mathcal{R}_2) \\
&\subseteq \mathcal{L}((\mathcal{R}_1\mathcal{R}_1^{\overline{*}}\mathcal{R}_2 \cap_{\not\epsilon} \mathcal{R}_2) \\
&\qquad \mid \texttt{M2-rec}(\mathcal{R}_1\mathcal{R}_1^{\overline{*}}\mathcal{R}_2, E \cup \{\mathcal{R}_1^*\mathcal{R}_2\}) \mid \mathcal{R}_1^* \cdot \texttt{M2-rec}(\mathcal{R}_2, E)) \\
&\qquad\qquad \text{(inductive hypothesis and } I \subseteq \mathcal{L}(\mathcal{R}_1\mathcal{R}_1^{\overline{*}}\mathcal{R}_2 \cap_{\not\epsilon} \mathcal{R}_2)) \\
&\subseteq \mathcal{L}(\mathcal{R}_1\mathcal{R}_1^{\overline{*}}\mathcal{R}_2 \cap_{\not\epsilon} \mathcal{R}_2) \cup \mathcal{M}_2(\mathcal{R}_1\mathcal{R}_1^{\overline{*}}\mathcal{R}_2) \cup \mathcal{M}_2(\mathcal{R}_2) \qquad \text{(inductive hypothesis)} \\
&= \mathcal{M}_2(\mathcal{R}_1^*\mathcal{R}_2) \qquad \text{(analogous to subcase } \mathcal{R} = \mathcal{R}_1^*\mathcal{R}_2 \text{ if } E = \emptyset)
\end{aligned}
$$

The overall correctness of `M2` (Thm. 4.1) is then a corollary of the correctness of `M2-rec` (Thm. A.1).

Corollary A.1 (Correctness of `M2`)
Let $\mathcal{R} \in \mathbb{R}$.

$$\mathcal{L}(\texttt{M2}(\mathcal{R})) = \mathcal{M}_2(\mathcal{R})$$

Proof
Follows immediately from the fact that $\texttt{M2}(\mathcal{R})$ is $\texttt{M2-rec}(\mathcal{R}, \emptyset)$. The precon-

dition of Thm. A.1 holds for the arguments. By applying Thm. A.1, we can observe what follows.

$$\mathcal{L}(\texttt{M2}(\mathcal{R})) = \mathcal{L}(\texttt{M2-rec}(\mathcal{R}, \emptyset)) = \mathcal{M}_2(\mathcal{R})$$

We now prove the correctness of Lemma 4.2.

Proof (Lemma 4.2)
Let t' be the subtree from the root $(\mathcal{R}, w)$ to the nodes in the frontier $\texttt{front}((\mathcal{R}, w))$. Observe that the nodes in $\texttt{front}((\mathcal{R}, w))$ are the only ones that possibly have subtrees outside the portion that we are considering: all the others are either internal nodes in t' or do not have children. Observe also that the number of nodes in t' does not depend on $|w|$, but just on $\mathcal{R}$ and the first character of w, if there is any.
The number of nodes in $\texttt{front}((\mathcal{R}, w))$ is bounded by $|\texttt{rch}(\mathcal{R})|$, since there is at most one occurrence of any regex $\mathcal{R}_1 \in \texttt{rch}(\mathcal{R})$ in $\texttt{front}((\mathcal{R}, w))$. This is because, if there were two occurrences of any $\mathcal{R}_1 \in \texttt{rch}(\mathcal{R})$, this would violate the hypothesis $\mathcal{M}_2(\mathcal{R}) = \emptyset$: there would be two different traces to match the first character of w. Furthermore, for the same reason, it holds that for each $i \in \{1, \ldots, |w|\}$:

$$|\texttt{front}^i((\mathcal{R}, w))| \leq |\texttt{rch}(\mathcal{R})|$$

Since the width of the matching tree grows as the size of the frontiers, this implies that the width of the matching tree is $\mathcal{O}(|\texttt{rch}(\mathcal{R})|)$. By Lemma 4.1, the height of the matching tree is at most linear in the length of the word, so that $|[\![\mathcal{R}]\!](w)| = \mathcal{O}(|w| \cdot |\texttt{rch}(\mathcal{R})|)$. Since $|\texttt{rch}(\mathcal{R})|$ does not depend on $|w|$, we conclude that $|[\![\mathcal{R}]\!](w)| = \mathcal{O}(|w|)$.

We can finally prove the soundness of our analysis (Thm. 4.2).

Proof (Thm. 4.2)
We prove the theorem by induction on $\texttt{nstars}(\mathcal{R})$. The base case is $\texttt{nstars}(\mathcal{R}) = 0$, namely in $\mathcal{R}$ there are no stars. We observe that stars are the only constructors that allow matching an arbitrary number of characters, which implies that the size of each matching tree is bounded by a constant

that does not depend on the input word, namely $|[\![\mathcal{R}]\!](w)| = \mathcal{O}(1)$. This can be seen as a consequence of the fact that $\mathcal{L}(\mathcal{R})$ is finite.
The inductive case is $\texttt{nstars}(\mathcal{R}) \geq 1$. The only case that we consider is when $\texttt{regex-head}(\mathcal{R}) = \mathcal{R}_1^*$ and $\texttt{regex-tail}(\mathcal{R}) = \mathcal{R}_2$. All other cases can be reduced to this: constructors that are not stars can match only a constant number of characters before reaching a star. Observe that by definition of $\mathcal{E}$, $\mathcal{E}(\mathcal{R}_1^*\mathcal{R}_2, \epsilon, \epsilon) =_{\mathcal{L}} \bot_r$ implies $\mathcal{E}(\mathcal{R}_2, \mathcal{R}_1^*, \epsilon) =_{\mathcal{L}} \bot_r$. Since the prefixes do not change the emptiness of the result, $\mathcal{E}(\mathcal{R}_2, \epsilon, \epsilon) =_{\mathcal{L}} \bot_r$. By observing that $\texttt{nstars}(\mathcal{R}_2) < \texttt{nstars}(\mathcal{R}_1^*\mathcal{R}_2)$, we can apply the inductive hypothesis and obtain the following.

$$|[\![\mathcal{R}_2]\!](w)| = \mathcal{O}(|w|^{\texttt{nstars}(\mathcal{R}_2)}) = \mathcal{O}(|w|^{\texttt{nstars}(\mathcal{R}_1^*\mathcal{R}_2)-1})$$

Therefore, for all $w \in \Sigma^*$, if w' is a suffix of w, all subtrees $[\![\mathcal{R}_2]\!](w')$ of $[\![\mathcal{R}]\!](w)$ have size at most polynomial in $|w'|$, which implies that the size is at most polynomial in $|w|$. Since $\mathcal{E}(\mathcal{R}_1^*\mathcal{R}_2, \epsilon, \epsilon) =_{\mathcal{L}} \bot_r$, then $\texttt{M2}(\mathcal{R}_1^*) =_{\mathcal{L}} \bot_r$. By Lemma 4.2 we can observe that matching any word in $\mathcal{R}_1^*$ is at most linear in the length of the input word. Let $w \in \Sigma^*$. In $[\![\mathcal{R}_1^*\mathcal{R}_2]\!](w)$ there are at most $|w|$ nodes of type $(\mathcal{R}_2, w')$ after matching any prefix of w in $\mathcal{R}_1^*$, namely at most one for any prefix of w. This is because $\texttt{M2}(\mathcal{R}_1^*) =_{\mathcal{L}} \bot_r$ implies that it is not possible to have two different traces that match any prefix of w. These observations imply that the matching tree can be decomposed in the part in which $\mathcal{R}_1^*$ is expanded (which is linear), and at most $|w|$ subtrees in which $\mathcal{R}_2$ is expanded. We already observed that all those subtress have size $\mathcal{O}(|w|^{\texttt{nstars}(\mathcal{R}_1^*\mathcal{R}_2)-1})$. Therefore, we obtain:

$$\begin{aligned}
|[\![\mathcal{R}_1^*\mathcal{R}_2]\!](w)| &= \mathcal{O}(|w|) + \sum_{i=1}^{|w|} \mathcal{O}(|w|^{\texttt{nstars}(\mathcal{R}_1^*\mathcal{R}_2)-1}) \\
&= \mathcal{O}(|w|) + |w|\mathcal{O}(|w|^{\texttt{nstars}(\mathcal{R}_1^*\mathcal{R}_2)-1}) \\
&= \mathcal{O}(|w|^{\texttt{nstars}(\mathcal{R}_1^*\mathcal{R}_2)})
\end{aligned}$$

This proves the theorem. Observe that actually $\mathcal{E}(\mathcal{R}_1^*\mathcal{R}_2, \epsilon, \epsilon) =_{\mathcal{L}} \bot_r$ can be caused not exclusively by $\texttt{M2}(\mathcal{R}_1^*) =_{\mathcal{L}} \bot_r$, but also by $\overline{\mathcal{R}_1^*\mathcal{R}_2} =_{\mathcal{L}} \bot_r$. The only

language that has as complement the empty language is Σ^*, which implies that $\mathcal{L}(\mathcal{R}_1^*\mathcal{R}_2) = \Sigma^*$. This case is then analogous to the previous one, because even though there might be an exponential number of traces to match a word in $\mathcal{R}_1^*$, only one is actually expanded, since $\mathcal{R}_1^*\mathcal{R}_2$ accepts any word. In this case, there exists no suffix that can make the match fail and trigger the exhaustive exploration of the set of traces.

Proofs for Chapter 7

Proof (Equation (7.14))
Let $\mathscr{R} \in \wp(\mathbb{D})$ and $T \in \wp(\mathbb{V})$.

$$\begin{aligned}
\alpha_t(\mathscr{R}) \supseteq T &\iff T \subseteq \alpha_t(\mathscr{R}) \\
&\iff T \subseteq \{\, \mathtt{x} \in \mathbb{V} \mid \mathscr{R} \subseteq \mathscr{T}(\mathtt{x}) \,\} \\
&\iff \forall \mathtt{x} \in T : \mathscr{R} \subseteq \mathscr{T}(\mathtt{x}) \\
&\iff \forall \mathtt{x} \in T : \forall R \in \mathscr{R} : R \in \mathscr{T}(\mathtt{x}) \\
&\iff \forall R \in \mathscr{R} : \forall \mathtt{x} \in T : R \in \mathscr{T}(\mathtt{x}) \\
&\iff \mathscr{R} \subseteq \{\, R \mid \forall \mathtt{x} \in T : R \in \mathscr{T}(\mathtt{x}) \,\} \\
&\iff \mathscr{R} \subseteq \bigcap_{\mathtt{x} \in T} \mathscr{T}(\mathtt{x}) \\
&\iff \mathscr{R} \subseteq \gamma_t(T)
\end{aligned}$$

Before proving Thm. 7.1, we give an alternative characterization of $\mathscr{NE}$.

$$\mathscr{NE} = \{\, R \in \mathbb{D} \mid \mathtt{ret} \notin \alpha_t(\{\, R \,\}) \,\} \tag{A1}$$

Proof

$$\begin{aligned}
&\mathscr{NE} \\
&= \{R \in \mathbb{D} \mid \forall((m_0, i_0, r_0), (m_1, i_1, r_1)), ((m_0', i_0', r_0'), (m_1', i_1', r_1')) \in R :
\end{aligned}$$

$$m_0 = m'_0, i_0 \neq i'_0, r_0 = r'_0 \implies m_1(\texttt{ret}) = m'_1(\texttt{ret})\}$$

(definition of $\mathscr{NE}$)

$$= \{R \in \mathbb{D} \mid \nexists((m_0, i_0, r_0), (m_1, i_1, r_1)), ((m'_0, i'_0, r'_0), (m'_1, i'_1, r'_1)) \in R :$$
$$m_0 = m'_0, r_0 = r'_0 : m_1(\texttt{ret}) \neq m'_1(\texttt{ret})\}$$ (negation of $\forall$)

$$= \{R \in \mathbb{D} \mid \texttt{ret} \in \overline{\alpha_t(\{R\})}\}$$ (definition of α_t)

$$= \{R \in \mathbb{D} \mid \texttt{ret} \notin \alpha_t(\{R\})\}$$ (α_t defines a partition over $\mathbb{V}$)

Proof (Thm. 7.1)
Follows immediately from Equation (A1).

Proof (Equation (7.15))

$$\begin{aligned}
&\alpha_t(\{R_0\}) \\
&= \{\,\mathtt{x} \in \mathbb{V} \mid \{R_0\} \subseteq \mathscr{T}(\mathtt{x})\,\} \\
&= \{\,\mathtt{x} \in \mathbb{V} \mid R_0 \in \mathscr{T}(\mathtt{x})\,\} \\
&= \{\,\mathtt{x} \in \mathbb{V} \mid R_0 \in \{\,R \in \mathbb{D} \mid \\
&\qquad \exists((m_0, i_0, r_0), (m_1, i_1, r_1)), ((m'_0, i'_0, r'_0), (m'_1, i'_1, r'_1)) \in R : \\
&\qquad\quad m_0 = m'_0, r_0 = r'_0 : m_1(\mathtt{x}) \neq m'_1(\mathtt{x})\,\}\,\} \qquad \text{(definition of } \mathscr{T}(\mathtt{x})\text{)} \\
&= \{\,\mathtt{x} \in \mathbb{V} \mid \exists((m_0, i_0, r_0), (m_1, i_1, r_1)), ((m'_0, i'_0, r'_0), (m'_1, i'_1, r'_1)) \in R_0 : \\
&\qquad\quad m_0 = m'_0, r_0 = r'_0 : m_1(\mathtt{x}) \neq m'_1(\mathtt{x})\,\} \\
&\subseteq \{\,\mathtt{x} \in \mathbb{V} \mid \exists((m_0, i_0, r_0), (m_1, i_1, r_1)), ((m'_0, i'_0, r'_0), (m'_1, i'_1, r'_1)) \in R_1 : \\
&\qquad\quad m_0 = m'_0, r_0 = r'_0 : m_1(\mathtt{x}) \neq m'_1(\mathtt{x})\,\} \qquad (R_0 \subseteq R_1) \\
&= \alpha_t(\{R_1\})
\end{aligned}$$

Proof (Thm. 7.2)

$$\begin{aligned}
R_1 \in \mathscr{NE} &\iff \texttt{ret} \notin \{\,\mathtt{x} \mid R_1 \in \mathscr{T}(\mathtt{x})\,\} && \text{(Thm. 7.1)} \\
&\implies \texttt{ret} \notin \{\,\mathtt{x} \mid R_0 \in \mathscr{T}(\mathtt{x})\,\} && (R_0 \subseteq R_1 \text{ and Equation (7.15)}) \\
&\iff R_0 \in \mathscr{NE} && \text{(Thm. 7.1)}
\end{aligned}$$

Proof (Thm. 7.3)
By structural induction. Skip statements and statement composition are trivial, and we do not report them. The correctness of `x = input()` follows from the fact that the variable `x` is the only one that assumes a different value, and hence can become tainted after the execution of the statement. For this reason, $\{\, \mathtt{y} \in T \mid \mathtt{y} \not\equiv \mathtt{x} \,\}$ exactly corresponds to the set of tainted variables after the execution of the statement using the hypothesis $T = \alpha_t(\{\, R \,\})$. Using the definition of α_t, we can observe that `x` is tainted if and only if the first element in the sequence of input can be controlled by the user, namely there are two executions whose initial states differ only in the user input and end in different first values for the input sequence. Random read statements are analogous.
Assignments are similar. Again, the only variable that can assume a different value is `x`, so that it is the only one that can become possibly tainted. For this reason, $\{\, \mathtt{y} \in T \mid \mathtt{y} \not\equiv \mathtt{x} \,\}$ exactly corresponds to the set of other tainted variables after the execution of the statement using the hypothesis $T = \alpha_t(\{\, R \,\})$. Using the definition of α_t, we observe that `x` is tainted iff there are two executions that differ in the initial states only in the input sequence, and result in different (non-error) values for the arithmetic evaluation in the input memory. This exactly corresponds to our definition.
By using α_t, we can observe that the tainted variables after the execution of if statements are: 1) those tainted in the then branch; 2) those tainted in the else branch; 3) those that assume different values in the two branches, in case which one of the two branches is executed depends on the user input. The first two cases are simple, and follow by inductive hypothesis. Observe that $T \setminus \overline{\alpha_t(\{\mathtt{test}[\![\mathtt{B}]\!]R\})}$ exactly corresponds to the set of tainted variables in the then branch under our assumption that $T = \alpha_t(\{R\})$. The same holds for $T \setminus \overline{\alpha_t(\{\mathtt{test}[\![\neg\mathtt{B}]\!]R\})}$ and the else branch. The third case is covered by the definition of $\mathtt{diff}[\![\mathtt{if\ (B)\ S}_t\ \mathtt{else\ S}_e]\!]$, as it considers all possible pairs of execution that in the initial state differ only in the user input, explore different branches, and then result in different values for some variables. For while statements it is sufficient to observe that $\lambda(R_1, E_1, T_1)\,.\,(R, E, T) \dot{\cup} \hat{\mathcal{S}}_t[\![\mathtt{if\ (B)\ S}_b\ \mathtt{else\ skip}]\!](R_1, E_1, T_1)$ is monotonic, and

the correctness of the rule follows from the correctness for if statements.

Proof (Thm. 7.4)
By structural induction. Some cases are trivial, and we do not report them.
For input read statements, it is sound to always taint the variable `x`, which is the only variable that possibly assumes a different value.
For random read statements, the only variable that can assume a different value is `x`, so that `x` is the only variable that can become tainted after the execution of `x = rand()`. By considering the definition of $\hat{\mathcal{S}}_t[\![\texttt{x = rand()}]\!]$, we observe that this happens only if the user can control which number in the sequence of random numbers is read. The soundness then follows by inductive hypothesis observing that if the user can control the value of the index variable `i`, then `i` is tainted. If `i` is tainted, then we taint `x`.
Similarly to the previous cases, `x` is the only variable that can become tainted after the execution of assignment `x = A`. Then, the soundness of the abstract semantics follows from the fact that $\texttt{taint}^\sharp[\![\texttt{A}]\!]$ returns `ff` only if the result of the arithmetic evaluation is definitely not influenced by user input.
For if statements, the fact that the tainted variables computed in the branches are an overapproximation of the truly tainted ones follows by inductive hypothesis. Then, we observe that $\texttt{diff}^\sharp[\![\texttt{if (B)}\ \texttt{S}_t\ \texttt{else}\ \texttt{S}_e]\!]$ returns more variables than $\texttt{diff}[\![\texttt{if (B)}\ \texttt{S}_t\ \texttt{else}\ \texttt{S}_e]\!]$. If a variable `x` is in $\texttt{diff}[\![\texttt{if (B)}\ \texttt{S}_t\ \texttt{else}\ \texttt{S}_e]\!]$, then the user can control the outcome of $\mathcal{B}[\![\texttt{B}]\!]$, and `x` is definitely assigned in at least one of the branches. We can then conclude by using the soundness of $\texttt{assigned}^\sharp[\![\texttt{S}]\!]$ and the fact that if the user can control the evaluation of the boolean condition, then $\texttt{taint}^\sharp[\![\texttt{B}]\!]$ is `tt`. Since while statements are defined in terms of if statements, the soundness of the former follows from the soundness of the latter.

The following result is useful to observe the connection between the $\mathcal{S}_t[\![\texttt{S}]\!]$ and $\mathcal{S}_t^\sharp[\![\texttt{S}]\!]$.

$$\alpha_t(\{\gamma_d(R^\sharp)\}) \subseteq T^\sharp \implies \mathcal{S}_t[\![\texttt{S}]\!](\gamma_d(R^\sharp), \gamma_d(E^\sharp)) \dot{\subseteq} \gamma(\mathcal{S}_t^\sharp[\![\texttt{S}]\!](R^\sharp, E^\sharp, T^\sharp)) \tag{A2}$$

Proof

$$\begin{aligned}
\mathcal{S}_t[\![\mathtt{S}]\!](\gamma_d(R^\sharp), \gamma_d(E^\sharp)) &\doteq \hat{\mathcal{S}}_t[\![\mathtt{S}]\!](\gamma_d(R^\sharp), \gamma_d(E^\sharp), \alpha_t(\{\gamma_d(R^\sharp)\})) && \text{(Thm. 7.3)}\\
&\dot{\subseteq} \hat{\mathcal{S}}_t[\![\mathtt{S}]\!](\gamma_d(R^\sharp), \gamma_d(E^\sharp), T^\sharp)\\
&\qquad (\alpha_t(\{\gamma_d(R^\sharp)\}) \subseteq T^\sharp \text{ and monotonicity of } \hat{\mathcal{S}}_t[\![\mathtt{S}]\!])\\
&\dot{\subseteq} \gamma(\mathcal{S}^\sharp_t[\![\mathtt{S}]\!](R^\sharp, E^\sharp, T^\sharp)) && \text{(Thm. 7.4)}
\end{aligned}$$

Proof (Thm. 7.5)
The fundamental observation to prove that if `ret` is tainted in the concrete semantics, then it is tainted in the abstract semantics is that if `ret` is tainted in the concrete, then there is *at least* one statement in which a runtime error occurs, and such an error can be triggered by the user. Since the abstract semantics taints `ret` every time there is a *possible* runtime error that could be triggered by the user, if `ret` is tainted in the concrete semantics, it will definitely be tainted in the abstract semantics.

Appendix B

RAT Implementation Details

We implemented our ReDoS analysis detection technique in the RAT [40] tool (**R**eDoS **A**bstract **T**ester) in less than 5000 lines of OCaml code. In this section, we describe the most meaningful portions of the implementation. *Transitional regular expressions* (see Section 4.3), which are the input of the analysis, are represented as an algebraic data type. The stars are labelled as *closed* or not by a boolean flag. Transitional regular expressions are implemented in the `Re` module.

```
(** Type of the regular expressions. *)
type t =
  | Epsilon
  | Char of Charset.t
  | Concat of t * t
  | Alternative of t * t
  | Star of bool * t (* true if the star can be expanded. *)
```

Characters are implemented as *character classes*, namely non-empty sets of characters rather than single characters. As discussed in Section 5.4, this enhances the performance of our implementation. The output of our ReDoS analysis is a possibly-empty regular expression, namely an element of $\mathbb{R}^{\perp}$. We implement $\mathbb{R}^{\perp}$ as *extended* regular expressions (see Section 3.2), namely possibly empty regular expressions with symbolic intersection and complement operations. Since in our algorithm we extensively use such operations, this again improves the performance of our detector. The extended regular expressions are implemented in the `ExtRe` module as follows.

```
(** Extended regular expressions. *)
```

```
type t =
  | Empty
  | Epsilon
  | Char of Charset.t
  | Concat of t list
  | Alternative of t list
  | Star of t
  | Inter of t list
  | Compl of t
```

The concatenation, alternative, and intersection constructors use lists rather than pairs of regular expressions. This representation makes it more efficient to perform some operations, such as *folding*, on large expressions. The `ExtRe` module exposes *smart constructors* (see Section 3.2), namely constructors that automatically reduce the size of the regular expressions when building them, while preserving the accepted language. An example is `concat`, which automatically removes `Epsilon` from the concatenation. To convert from `Re.t` (i.e., transitional regular expressions) to `ExtRe.t` (i.e., extended regular expressions), we use the `to_ext_re` function, which is trivial and we do not report here.

Observe that we do not use smart constructors and optimized algorithms to build transitional regular expressions, as it is important to faithfully represent the input of the analysis. This is due to the fact that we do not want to inject or remove potential ReDoS vulnerabilities, which depend on the *syntax* of the regular expressions. Consider, for instance, the vulnerable expression $(a \mid a)^*$. If we used a smart constructor for the alternative that simplifies $\mathcal{R} \mid \mathcal{R}$ to $\mathcal{R}$, we would obtain as input a^*, where the vulnerability has been removed. This would result in an unsound analysis. On the other hand, extended regular expressions are used only to compute the attack language, and can leverage smart constructors in order to reduce their size and, therefore, improve the performance.

The attack language computed by our analysis is the union of regular expressions that have the form $\mathcal{P} \cdot \mathcal{R}^* \cdot \mathcal{S}$, for some prefix $\mathcal{P}$, pump $\mathcal{R}$, and suffix $\mathcal{S}$. We represent this regular expression with a specific type in the `AttackFamily` module:

```
type t = {
  prefix : ExtRe.t;
  pump   : ExtRe.t;
  suffix : ExtRe.t
```

```
}
```

The result of the analysis is then a set of `AttackFamily.t`, which in our implementation is `AttackFamilySet.t`. The analysis matches the definition of $\mathcal{E}$, and it is implemented as follows.

```
(** [exp_attack_families r] returns the families of
    exponentially attack words for the regex [r]. *)
let rec exp_attack_families r =
  exp_attack_rec ExtRe.eps ExtRe.eps r
    |> AttackFamilySet.remove_empty

and exp_attack_rec pref suff r =
  match r with
  | Epsilon -> AttackFamilySet.empty
  | Char _ -> AttackFamilySet.empty
  | Alternative (r1, r2) ->
    exp_attack_rec_alternative pref suff r1 r2
  | Concat (r1, r2) ->
    exp_attack_rec_concat pref suff r1 r2
  | Star (_, r1) ->
    exp_attack_rec_star pref suff r r1

and exp_attack_rec_alternative pref suff r1 r2 =
  AttackFamilySet.union
    (exp_attack_rec pref suff r1)
    (exp_attack_rec pref suff r2)

and exp_attack_rec_concat pref suff r1 r2 =
  AttackFamilySet.union
  (exp_attack_rec pref (ExtRe.concat (to_ext_regex r2) suff) r1)
  (exp_attack_rec (ExtRe.concat pref (to_ext_regex r1)) suff r2)

and exp_attack_rec_star pref suff r r1 =
  let pref = ExtRe.concat pref (to_ext_regex r) in
  let suff = ExtRe.concat (to_ext_regex r) suff in
  let negated_suff = ExtRe.compl suff in
  let pump = m2 r in
  let attack_family =
    AttackFamilySet.singleton
      { prefix = pref; pump; suffix = negated_suff }
```

```
  in
  let attack_e' = exp_attack_rec pref suff r1 in
  AttackFamilySet.union attack_family attack_e'
```

The function `exp_attack_rec` exactly corresponds to the analysis $\mathcal{E}$. We now show how the function `M2` (see Algorithm 2) is implemented. The module `RS` is an alias to the **R**egex**S**et module, that implements sets of regular expressions.

```
(** [m2 r] returns the language of words that can possibly be
    matched in at least two traces in the expression [r]. *)
let rec m2 r = m2_rec r RS.empty

and m2_rec r explored =
  if RS.mem r explored then ExtRe.empty
  else
    match (head r, tail r) with
    | Epsilon, _ | Star (false, _), _ -> ExtRe.empty
    | (Char _ as a), r1 ->
      ExtRe.concat
        (to_ext_regex a)
        (m2_rec (Re.refesh_stars r1) explored)
    | Alternative (r1, r2), r3 ->
      let inter =
        non_eps_iter (Concat (r1, r3)) (Concat (r2, r3)) in
      let left = m2_rec (Concat (r1, r3)) explored in
      let right = m2_rec (Concat (r2, r3)) explored in
      ExtRe.alternative inter (ExtRe.alternative left right)
    | (Star (true, r1) as r1_star), r2 ->
      let expanded =
        Concat (r1, Concat (Star (false, r1), r2)) in
      let inter = non_eps_iter expanded r2 in
      let left = m2_rec expanded (RS.add r explored) in
      let right =
        ExtRe.concat
          (to_ext_regex r1_star) (m2_rec r2 explored) in
      ExtRe.alternative inter (ExtRe.alternative left right)
```

The last interesting part of the code that we report is the procedure `non_eps_inter`, which computes the $\cap_{\not\epsilon}$ operator as described in Section 4.4 (see Algorithm 3).

```
(** [remove_eps r] returns a regular expression [r'] that
    accepts the same language as [r] without [Epsilon]. *)
```

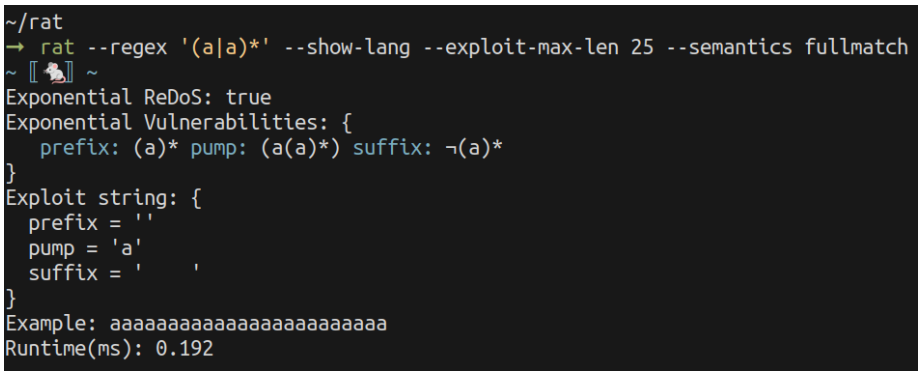


FIGURE A1. Example usage of the RAT tool

```
3  let rec remove_eps r =
4    match (head r, tail r) with
5    | Epsilon, _ | Star (false, _), _ -> ExtRe.empty
6    | (Char _ as a), r1 ->
7      ExtRe.concat
8        (to_ext_regex a) (refesh_stars r1 |> to_ext_regex)
9    | Alternative (r1, r2), r3 ->
10     ExtRe.alternative
11       (remove_eps (Concat (r1, r3)))
12       (remove_eps (Concat (r2, r3)))
13   | Star (true, r1), r2 ->
14     ExtRe.alternative
15       (remove_eps (Concat (r1, Concat (Star (false, r1), r2))))
16       (remove_eps r2)
17
18 let non_eps_iter r1 r2 =
19   ExtRe.inter (remove_eps r1) (remove_eps r2)
```

Figure A1 shows RAT's output when analyzing the regular expression $(a \mid a)^*$ assuming the fullmatch semantics. Observe that the tool prints the computed attack language and an example exploit string. The length of the exploit string can be adjusted with a command-line parameter. Exploit strings are obtained by transforming the attack language into an automaton (see Section 3.3 for an overview of the existing conversion methods between regular expressions and automata), and then performing a breadth-first search from the initial state to any accepting state to compute an attack word.

Appendix C

Interval analysis helper functions

In this section, we define the helper functions used in the analysis proposed in Section 7.8. First, by relying on $\texttt{zero}^\sharp[\![\texttt{A}]\!]$, we define a function that determines whether there is a possible runtime error in the evaluation of expressions:

$$\begin{aligned}
\texttt{haserror}^\sharp[\![\texttt{A}]\!]\bot^\sharp &\triangleq \texttt{ff}\\
\texttt{haserror}^\sharp[\![\texttt{n}]\!]R^\sharp &\triangleq \texttt{ff}\\
\texttt{haserror}^\sharp[\![\texttt{x}]\!]R^\sharp &\triangleq \texttt{ff}\\
\texttt{haserror}^\sharp[\![\texttt{A}_1 \diamond \texttt{A}_2]\!]R^\sharp &\triangleq \begin{cases} \texttt{tt} & \text{if } \diamond = /,\ \texttt{zero}^\sharp[\![\texttt{A}_2]\!]R^\sharp \\ \texttt{tt} & \text{if } \texttt{haserror}^\sharp[\![\texttt{A}_1]\!]R^\sharp \text{ or } \texttt{haserror}^\sharp[\![\texttt{A}_2]\!]R^\sharp \\ \texttt{ff} & \text{otherwise} \end{cases}
\end{aligned}$$

The function $\texttt{assigned}^\sharp[\![\texttt{S}]\!]$ must be sound with respect to the following:

$$\begin{aligned}
\{\,\texttt{x} \mid\ & \exists((m_0, i_0, r_0), (m_1, i_1, r_1)) \in \gamma_t(R^\sharp) : \exists(m_2, i_2, r_2) :\\
& ((m_0, i_0, r_0), (m_2, i_2, r_2)) \in \mathcal{S}[\![\texttt{S}]\!](\{((m_0, i_0, r_0), (m_1, i_1, r_1))\}, \emptyset) :\\
& m_1(\texttt{x}) \neq m_2(\texttt{x})\,\} \subseteq \texttt{assigned}^\sharp[\![\texttt{S}]\!]R^\sharp
\end{aligned}$$

Then, for intervals, we define the function $\texttt{assigned}_i^\sharp[\![\texttt{S}]\!] : \mathbb{V}_\mathbb{I}^\sharp \to \wp(\mathbb{V})$ as follows.

$$
\begin{aligned}
\texttt{assigned}_i^\sharp[\![\texttt{S}]\!]\bot^\sharp &\triangleq \emptyset \\
\texttt{assigned}_i^\sharp[\![\texttt{skip}]\!]R^\sharp &\triangleq \emptyset \\
\texttt{assigned}_i^\sharp[\![\texttt{S}_1\,\texttt{;}\,\texttt{S}_2]\!]R^\sharp &\triangleq \texttt{assigned}_i^\sharp[\![\texttt{S}_1]\!]R^\sharp \cup \texttt{assigned}_i^\sharp[\![\texttt{S}_2]\!](\mathcal{S}^\sharp_{\mathbb{V}_\mathbb{I}^\sharp}[\![\texttt{S}_1]\!]R^\sharp) \\
\texttt{assigned}_i^\sharp[\![\texttt{x = input()}]\!]R^\sharp &\triangleq \{\texttt{x}\} \\
\texttt{assigned}_i^\sharp[\![\texttt{x = rand()}]\!]R^\sharp &\triangleq \{\texttt{x}, \texttt{i}\} \\
\texttt{assigned}_i^\sharp[\![\texttt{x = A}]\!]R^\sharp &\triangleq \{\,\texttt{ret} \mid \texttt{haserror}^\sharp[\![\texttt{A}]\!]R^\sharp\,\} \cup \\
&\qquad \{\,\texttt{x} \mid \neg\texttt{isconst}_i^\sharp[\![\texttt{A}]\!]R^\sharp \text{ or } \neg\texttt{isconst}_i^\sharp[\![\texttt{x}]\!]R^\sharp \text{ or} \\
&\qquad\qquad R^\sharp(\texttt{x}) \neq \mathcal{A}^\sharp_{\mathbb{V}_\mathbb{I}^\sharp}[\![\texttt{A}]\!]R^\sharp\,\} \\
\texttt{assigned}_i^\sharp[\![\texttt{if (B) S}_t\,\texttt{else S}_e]\!]R^\sharp &\triangleq \{\,\texttt{ret} \mid \texttt{haserror}^\sharp[\![\texttt{B}]\!]R^\sharp\,\} \cup \\
&\qquad \texttt{assigned}_i^\sharp[\![\texttt{S}_t]\!](\texttt{test}^\sharp_{\mathbb{V}_\mathbb{I}^\sharp}[\![\texttt{B}]\!]R^\sharp) \cup \\
&\qquad \texttt{assigned}_i^\sharp[\![\texttt{S}_e]\!](\texttt{test}^\sharp_{\mathbb{V}_\mathbb{I}^\sharp}[\![\neg\texttt{B}]\!]R^\sharp) \\
\texttt{assigned}_i^\sharp[\![\texttt{while (B) S}_b]\!]R^\sharp &\triangleq \textbf{let } (R_f^\sharp, X_f) = \lim F^n(\bot^\sharp, \emptyset) \textbf{ in } X_f \\
\textbf{where } F(R_1^\sharp, X_1) &\triangleq \textbf{let } R_2^\sharp = R_1^\sharp \,\nabla_i\, (R^\sharp \cup^\sharp_{\mathbb{V}_\mathbb{I}^\sharp} \mathcal{S}^\sharp_{\mathbb{V}_\mathbb{I}^\sharp}[\![\texttt{if (B) S}_b\,\texttt{else skip}]\!]R_1^\sharp) \textbf{ in} \\
&\qquad \textbf{let } X_2 = X_1 \cup \texttt{assigned}_i^\sharp[\![\texttt{if (B) S}_b\,\texttt{else skip}]\!]R_1^\sharp \textbf{ in} \\
&\qquad (R_2^\sharp, X_2)
\end{aligned}
$$

Observe that we do not include in $\texttt{assigned}_i^\sharp[\![\texttt{x = A}]\!]$ the variable x is case its value is not modified by the statement, for instance in x = 0 when x is already 0. For x not to be included in $\texttt{assigned}^\sharp[\![\texttt{x = A}]\!]$, the arithmetic evaluation of A must be constant, the previous value of x in $R^\sharp$ must be constant, and the two must be exactly the same interval.

List of Figures

List of Tables

List of Definitions, Theorems, Lemmas, and Corollaries

List of Examples