BRNO UNIVERSITY OF TECHNOLOGY

Faculty of Information Technology

Department of Intelligent Systems

Ing. Lenka Turoňová

Automata with Counting in Regular Expression Matching

Čítačové automaty ve vyhledávání podle regulárních výrazů

PH.D. THESIS

Supervisor: doc. Mgr. Lukáš Holík, Ph.D.
Co-Supervisor: prof. Ing. Tomáš Vojnar, Ph.D.

Abstract

Matching of *regular expressions* (regexes) is widely used, e.g., for searching, data validation, parsing, finding and replacing, data scraping, or syntax highlighting in many programming languages. It is a computationally intensive process often applied on large texts. Predictability of its efficiency has a significant impact on the overall usability of software applications in practice. A problem is that standard approaches for regex matching suffer from high worst case complexity. An unlucky combination of a regex and text may increase the matching time by orders of magnitude. This can be a doorway for the so-called *Regular expression Denial of Service* (ReDoS) attack in which the attacker causes a denial of service by providing a specially crafted regex or text.

Automata-based matchers are the most efficient regex matching engines used nowadays in practice, especially in performance-critical industrial applications. There are years of empirical evidence showing that their performance is much more stable than that of the more traditional backtracking-based matchers. But automata matchers may run into troubles too. Bounded repetition, i.e., expressions such as `[ab]{100}` with a specified number of repetitions of a certain pattern, has been recognised as a major source of problems for even the fastest matchers. This thesis studies this issue systematically.

In this thesis, we present a large-scale study of vulnerability of automata-based matching focused on bounded repetition. To this end, we propose a new ReDoS generator. It is the first generator capable of utilising bounded repetition to attack automata-based matchers, in fact the first generator that can attack them at all. We were then able to prove experimentally that bounded repetition indeed poses a serious security threat, for automata-based as well as backtracking-based matchers.

We then propose a solution to the problem of efficient matching of regexes with bounded repetition. The approach is to compile the regexes into nondeterministic *counting automata* (CAs) and then to determinise them. The main problem is to find a succinct deterministic representation that can perform fast matching (naive determinisation builds a *deterministic finite automata* (DFAs) exponentially large to the size of the regex and of the repetition bounds in it). In the first step, we propose a determinisation algorithm based on general subset construction that generates deterministic CAs. They are exponentially more succinct than the corresponding DFAs. The main contribution of this thesis was then obtained when we elaborated the determinisation using the idea representing many counters with *counting sets*. We propose succinct transformation of a CA into a deterministic *counting-set automaton* (CsA), an automaton with a special type of registers that can hold a set of integer values. We also propose a novel compilation of regexes to CAs that generalizes the Antimirov's derivative construction. We design a framework for matching based on CsA simulation and the Antimirov's derivative construction. We compare the speed of matching of individual matching engines on a comprehensive set of real-world regexes with bounded repetition. We found that our algorithm is much more robust, outperforms the state-of-the-art matchers on regexes with bounded repetition, and is not dependent on the size of repetition bounds. It easily solves most cases in which the existing matchers struggle due to bounded repetition.

Keywords

Regular expression matching, bounded repetition, ReDoS, determinisation, Antimirov's derivatives, counting automata, counting-set automata.

Abstrakt

Vyhledávání podle regulárních výrazů (regexové vyhledávání) je široce využívaný prostředek např. pro vyhledávání informací, ověřování dat, vyhledávání a nahrazování, získávání dat nebo zvýrazňování syntaxe v mnoha programovacích jazycích. Jedná se o výpočetně náročný proces, který se často aplikuje na rozsáhlé texty. Jeho předvídatelnost a stabilita má v praxi významný dopad na celkovou použitelnost softwarových aplikací. Problémem je, že standardní přístupy pro regexové vyhledávání mají vysokou složitost a nešťastná kombinace regexu a textu může dobu vyhledávání řádově prodloužit. To může být vstupní branou pro tzv. ReDoS útoky, což je závažný bezpečnostní problém, kdy útočník způsobí odepření služby pomocí speciálně vytvořeného regexu nebo textu.

Automatové regexové vyhledávače jsou v současné době nejefektivnějšími nástroji pro regexové vyhledávání používanými v praxi, zejména v průmyslových výkonnostně kritických aplikacích. Dlouholeté empirické studie ukazují, že tyto přístupy mají mnohem stabilnější výkonnost, než jakou mají existující nástroje pro regexové vyhledávání založené na zpětném prohledávání. Nicméně i automatové regexové vyhledávače se mohou dostat do potíží. Omezená opakování, např. `[ab]{100}`, představují hlavní zdroj problémů i pro nejrychlejší nástroje pro regexové vyhledávání. Tato práce se touto problematikou zabývá systematicky.

V této práci jsme nejprve představili rozsáhlou studii zranitelnosti nástrojů pro regexové vyhledávání založených na konečných automatech. Za tímto účelem jsme navrhli nový ReDoS generátor. Jedná se o první generátor schopný využívat omezené opakování ke generování útoků pro automatové regexové vyhledávače. Dále jsme byli schopni experimentálně prokázat, že omezená opakování skutečně představují vážnou bezpečnostní hrozbu, jak pro automatové regexové vyhledávače, tak pro ty založené na zpětném prohledávání.

Dále jsme navrhli řešení problému efektivního regexové vyhledávání s omezeným opakováním. Obecný přístup je založen na kompilaci regexů do nedeterministických čítačových automatů a jejich následné determinizaci. Hlavním problémem je najít stručnou deterministickou reprezentaci, která dokáže provádět rychlé regexové vyhledávání (naivní determinizace vytváří deterministické konečné automaty exponenciálně velké k velikosti regexu a k maximům mezí opakování, které se v nich nachází). Nejprve jsme navrhli determinizační algoritmus vycházející z klasické podmnožinové konstrukce, který generuje deterministické čítačové automaty. Tyto automaty jsou exponenciálně stručnější než odpovídající deterministické konečné automaty. Hlavní přínos této práce jsme pak získali, když jsme determinizaci rozpracovali pomocí myšlenky čítacích množin. Navrhli jsme stručnou transformaci čítačového automatu na deterministický automat se speciálním typem registrů, které mohou obsahovat množinu celočíselných hodnot. Představili jsme také novou kompilaci regexů na čítačové automaty, která zobecňuje Antimirovu derivatovou konstrukci. Vytvořili jsme aplikační rámec založený na simulaci automatů s čítačovými registry a Antimirově derivatové konstrukci. Porovnali jsme rychlost vyhledávání jednotlivých nástrojů na rozsáhlé sadě reálných regexů s omezeným opakováním. Zjistili jsme, že náš algoritmus je mnohem robustnější, překonává nejmodernější nástroje pro regexové vyhledávání na regexech

s omezeným opakováním a není závislý na velikosti mezí opakování. Snadno řeší většinu případů, ve kterých mají stávající nástroje pro regexové vyhledávání problém s omezeným opakováním.

Klíčová slova

Vyhledávání podle regulárních výrazů, omezené opakování, ReDoS, determinizace, Antimirovy derivativy, čítačové automaty.

Automata with Counting in Regular Expression Matching

Prohlášení

Prohlašuji, že jsem tuto disertační práci vypracovala samostatně pod vedením doc. Mgr. Lukáše Holíka, Ph.D. a prof. Ing. Tomáše Vojnara, Ph.D. Uvedla jsem všechny literární prameny a publikace, ze kterých jsem čerpala.

. . . . . .
Lenka Turoňová
12. května 2022

Acknowledgements

This work would not have been possible without the constant support, enthusiasm, guidance, encouragement, and assistance of my supervisor and life partner Lukáš Holík. His level of patience, knowledge, and ingenuity is something I will always keep aspiring to.

I wish to express my gratitude to Margus Veanes for his valuable suggestions, inspiring discussions, ever encouraging and motivating guidance, and the opportunity to do an internship in Microsoft Research. I also thank him for his patience and insightful feedback he gave me.

I am also grateful to my co-supervisor Tomáš Vojnar for his encouragement to start with Ph.D. studies in VeriFIT and for his support and financial means to complete my Ph.D. studies. I thank Ondřej Lengál and Ivan Homoliak for their efforts and contributions. I am also grateful for the opportunity to meet great people from our field, especially Parosh Aziz Abdulla and Yu-Fang Chen. My appreciation also goes to my family for their encouragement and support through my studies.

Contents

'When information overload occurs, pattern recognition is how to determine truth.'

Marshall McLuhan

1 Introduction

Managing large quantum of information is a problem faced by almost everyone. Social media, e-mails, etc. all spill data into our lives daily. To be able to make quick decisions, a person or a machine needs to identify patterns in text information.

One of the most widely used techniques is to use *regular expressions* (regexes). Regexes offer a sweet spot in the following sense. On one hand, they are simple, compact, human readable, and can be efficiently mechanized. On the other hand, they provide a good expressive power. Regex matching is therefore widely used, e.g., for searching, data validation, parsing, data scraping, or syntax highlighting, and also in less known applications, such as bioinformatics, where it is used to discover specified gene sequences which may cause particular diseases or to extract information from clinical reports [81]. It is natively supported in most programming languages [19]. For instance, about 30–40 % of Java, JavaScript, and Python software use regex matching (as reported in multiple studies, see, e.g., [25]). Regexes are often used in high-risk applications where it is critical to ensure stability, predictability, and quick evaluation, and avoid crashes and security vulnerabilities, e.g., in network processing or various security applications. For example, regexes are used to validate user input to prevent SQL injection attacks. Network intrusion detection systems deploy regex matching to monitor networks for malicious activity or policy violations.

Regex matching is a computationally intensive process often applied to large texts. Predictability of its efficiency has a significant impact on the overall usability of software applications. The worst case complexity of matching is high (super-linear to the text or exponential to the size of the regex, depending on the matching algorithm). This can be exploited by an attacker who can provide vulnerability-triggering text to increase the matching time by orders of magnitude. Such an attack is known as the so-called *ReDoS (regular expression denial of service) attack*. Unfortunately, developers are often unaware of security risks that can occur when using regexes. Very specific and rare texts may be needed to trigger an extreme behaviour and thus vulnerable regexes are easily missed by testing, and satisfactory analytical means for distinguishing vulnerable regexes do not exist.

The traditional backtracking matchers are considered most susceptible to ReDoS attacks. Performance critical applications are therefore using automata-based algorithms

which are indeed the most efficient regex matching engines used nowadays in practice, i.e., implemented in Google's `RE2`, `grep`, `SRM`, the standard matcher of `Rust`, or `Hyperscan` [42, 32, 82, 31, 54]. There are years of empirical evidence showing that their performance is much more stable than that of the backtracking-based matchers. However, automata matchers may run into troubles too. *Bounded repetition (or bounded quantifier/counting operator)*, i.e., used in expressions of such as '`[ab]{100}`', has been recognised as a major source of problems for even the fastest matchers. The running time of automata-based matchers is in fact linear to the repetition bounds. This makes matching with already moderately high quantifier bounds (in the order of hundreds) prone to significant slowdowns and ReDoS vulnerabilities. In practice, the repetition bounds are often high. In extreme, repetition bounds used in real life XML schemas may reach the order of tens of millions [10]. This thesis studies this issue systematically and proposes a solution.

1.1 Examples of ReDoS in History

As we already mentioned, regexes are often used in high-risk areas where vulnerable regexes represent a doorway for a ReDoS. Throughout the history, there have been numerous ReDoS attacks and their frequency is still on the rise [90] (e.g., in 2018, ReDoS exploits increased by 143 %). The following examples of recent incidents may serve as evidence of severity of this problem.

In 2019, Cloudflare has been experiencing a catastrophic outage caused by a web application firewall rule that contained a poorly written regex that ended up creating excessive backtracking and exhausted the CPU power used for HTTP/HTTPS serving [44]. The regex that caused the outage was '`(?:(?:\"|'|\]|\}|\\|\d|(?:nan|infinity|true|false|null|symbol|math) |\‘|\-|\+)+[)]*;?((?:\s|-|~|!|{}|\|\||\+)*.* (?:.*=.*)))`'. The key part of the vulnerable regex was '`.*.*=.*`' which seams to be a very simple regex, but for a backtracking matcher it takes 45 steps to match an input '`x=xxx`' and the number of steps grows super-linearly with an increasing number of '`x`' at the end.

In 2016, a malformed post with a vulnerable regex caused a high consumption of CPU on the StackOverflow web page [33]. The regex was '`^[ \s \u200c]+|[ \s \u200c]+$`', and it was intended to trim unicode spaces from the start and the end of the line. The simplified version of this regex, '`\s+$`', would expose the same issue. The post was in the homepage list and the vulnerable regex was thus called on each homepage view. The main issue was that a simple backtracking matcher had to match the regex against the malformed post containing around 20,000 consecutive whitespaces on a comment line that started with '– play happy sound for player to enjoy'. It caused a catastrophic backtracking which forced the matcher to perform a 199,990,000 steps. Thus, instead of a happy smile, the post caused horror among developers.

Even **express.js**, the most popular web framework for **node.js**, does not manage to eliminate the possibility of a ReDoS attack. In 2016, there was found a flaw in a module that is used in **express.js** [6]. The identified flaw relied on the express app using a feature called **acceptsLanguages()** commonly used for determining the preferred language of the client. Providing a specially crafted **Accept-Language** header, an attack could trigger catastrophic backtracking.

1.2 Existing Approaches to Pattern Matching

Let us now introduce the existing approaches to pattern matching. The most common approach is backtracking. Matching algorithms based on backtracking are used in the regex engines of wide-spread programming languages, e.g., `.NET` [65], `Python` [36], `Perl` [96], `PHP` [45], `Java` [30], `JavaScript` [21], or `Ruby` [11]. Backtracking is a recursive procedure that descends the syntactic tree of the regex while reading the text from the left to the right, matching its characters against subexpressions of the regex. The basic backtracking algorithm is simple and easily extensible with advanced features, however, it is at worst exponential in the text length. Regexes prone to extreme running times are easily constructed and found in practice [26]. ReDoS analyzers can often find triggering texts for regexes that are used in practice, and even some analytical methods for identifying regexes vulnerable to backtracking were proposed [109, 76, 107, 60].

The other major family of regular matching engines are automata-based matchers. A basic automata-based matching algorithm is the *offline DFA-simulation* [85]. In the ideal case, the DFA is pre-computed; matching can then be linear in the text length, with each input symbol processed in constant time. The major drawback of offline DFA-simulation is that the DFA construction may explode, rendering the method unusable in practice. Most of automata-based matchers that alleviate this problem evolved from the *Thompson's algorithm* [86] *aka the NFA-simulation*. In essence, the algorithm is a *breadth-first* exploration of the runs of the NFA along the input. In combination with caching, it becomes an on-the-fly subset construction of a part of the DFA needed to match the given word, also called the *online DFA-simulation*. Forms of the online DFA-simulation are implemented in the Google's `RE2` library [42], the standard GNU `grep` program [32], the `Rust` standard regex matcher [31], or the Symbolic Regex Matcher (`SRM`) [82].[1] Intel's `Hyperscan` [20] uses a variation of NFA-simulation algorithm as one of its components, among a number of other techniques.

The automata-based approaches are harder to implement, less flexible, and it is not clear how to extend them with advanced regex features such as back-references. On the other hand, they are more stable and less prone to ReDoS attacks and therefore overwhelmingly preferred when avoiding regex vulnerabilities is a priority. They are now prevailing in industrial applications such as network intrusion detection [63, 35] and credentials scanning [66].

1.3 Regexes with Bounded Repetition

As mentioned earlier, this thesis focuses on the Achilles heel of automata-based matchers, which are regexes with bounded repetition. Such expressions are very common in practice (cf. [10]), e.g., in the RegExLib library [78], which collects expressions for recognising URIs, markup code, pieces of Java code, or SQL queries; in the Snort rules [63] used for finding attacks in network traffic; or in real-life XML schemas, with the repetition bounds being as large as 10 million [10]. Repetition constraints may also naturally arise in other contexts, such as for automata-based verification approaches (e.g. [87]) for

[1] `SRM` is based on symbolic Antimirov's derivatives [4] constructed on the fly, also in the spirit of online DFA construction.

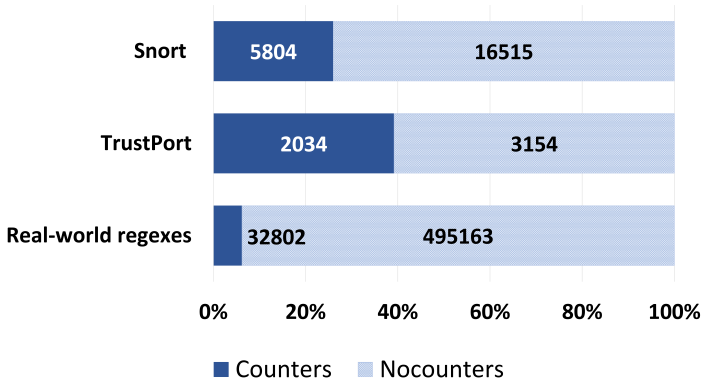


Figure 1.1: Numbers of regexes with/without bounded repetition in selected classes of regexes: Snort—network intrusion detection systems [63], TrustPort—detection of security breaches [97], and real-world regexes from software projects at GitHub [27].

describing runs with some number of iterations of a loop.

We analyzed over 500k real-world regexes obtained in the study performed by Davis et al. [27] and other sources. Figure 1.1 shows a ratio of regexes with/without bounded repetition in selected classes of regexes. Over 40k regexes out of the 500k above mentioned ones contain the bounded repetition.

Example 1.3.1. To illustrate the principal difficulty with matching bounded repetitions, especially when combined with a high degree of nondeterminism, consider the regex '`.*a.{`k`}`' where $k \in \mathbb{N}$ (the regex denotes strings where the symbol '`a`' appears k positions from the end of the word). Already the NFA will have at least k states, which is exponential in the regex size because k is written in decimal. Due to the inherent nondeterminism of this regex, determinisation then adds a second level of the exponential explosion. Indeed, the minimal DFA accepting the language has 2^{k+1} states because it must remember all the positions where the symbol '`a`' was seen during the last $k+1$ steps. This requires a finite memory of $k+1$ bits and thus 2^{k+1} reachable DFA states. Determinising the NFA explicitly is thus out of the question for even moderate values of k.

Generally, the worst-case complexity of a regex match depends on the size of the regex $|R|$ (resp. the size of the underlying automaton), as it is shown in Example 1.3.1, and length of the input text $|w|$. As w is normally supposed to be large and R small, $|w|$ is the more important parameter in the complexity. As we already mentioned, the worst-case complexity of matching using backtracking is exponential in $|w|$ and it is even higher if we use the regexes with bounded repetition.

Matching of regexes with bounded repetition is however problematic even for automata-based matchers. A simple way of translating such regexes to NFAs is to unfold the bounded repetition first and then use some of the well known regex-to-NFA-translations, such as Glushkov's, Thompson's, or Antimirov's algorithm [40, 86, 4]. These algorithms produce from a regex without bounded repetition an NFA with a linear number of states and roughly quadratic number of transitions. However, the unfolding of bounded repetition increases the size of the regex $\Theta((\boldsymbol{max}_R)^{cnt_R})$ times, with the worst-case occurring when R is a cnt_R-fold application of a bounded repetition with the upper bound $\boldsymbol{max}_R$ on some base regex S. Hence, the NFA for the regex with bounded

repetition has then $\Theta(|R|.(\boldsymbol{max}_R)^{cnt_R})$ states, or $\Theta(|R|.\boldsymbol{max}_R)$ states if R has *flat counting* (bounded repetition that is not nested), and quadratically as many transitions.

The offline DFA-simulation compiles R to NFA, determinises it, and follows the DFA run over w, all in time and space $\Theta(2^{|R|^2}+|w|)$. While the technique of online DFA simulation reduces the complexity to $\Theta(\min(2^{|R|^2}+|w|, |w|.|R|^2))$, where $|w|.|R|^2$ comes from that every step may incur construction of a new DFA state and transition in time $\Theta(|R|^2)$. The case of extended regexes is more problematic since the size of the underlying NFA depends at least linearly (in the case of flat counting) on $\boldsymbol{max}_R$ and thus exponentially on $|R|$. Hence, the worst-case complexity of DFA-simulation includes the factor $2^{(|R|.(\boldsymbol{max}_R)^{cnt_R})^2}$, double-exponential in the size of the regex (or worse in the case of nested counting). This makes matching with already moderately high repetition bounds (in the order of hundreds) prone to significant slowdowns and ReDoS vulnerabilities.

Finding a matching algorithm that would run in time independent of the repetition bounds has been an open problem for decades (as noted also in [92]). A number of proposals has been published [38, 10, 51, 56, 88, 55], but they are rather unsatisfactory, due to their cost super-linear in the length of the text (e.g. [55]) or exponential in repetition bounds (e.g. [88]) or due to a restricted class of supported regexes (e.g. [38]).

1.4 Existing Approaches to ReDoS Detection

The fact that ReDoS is indeed a common and serious threat is argued not only by our examples but also by several works such as [25, 26]. Therefore, stress-testing of regex matchers, one of the the topics of this thesis, is an active research area. Several methods and tools have been developed that attempt to determine whether a given regex is vulnerable to a ReDoS and to generate a triggering text.

The state-of-the-art ReDoS detectors [77, 107, 109, 83] focus only on regex matchers that are based on the backtracking algorithm. They search for regexes that cause the backtracking-based matching engines to run in super-linear time.

The detectors are either static, dynamic, or a combination of both based on whether actual regex matching is conducted. Static ReDoS generators (`RegexCheck` [109], `RegexStatic` [108]) are based purely on the analysis of an NFA obtained from a regex. They can be sound and complete for certain class of regexes but the major disadvantage of these detectors is that, in general, they suffer from poor accuracy.

Dynamic ReDoS generators (`SlowFuzz` [74], `RXXR2` [76, 77]) use some kind of evolutionary-search-based algorithms. They conduct actual regex matching and use the profiling results to improve next iterations of the algorithm. Thus, they report only true positive ReDoS. Moreover, they may handle regex extensions. The main drawback of the dynamic generators is that they do not scale and thus, they may miss ReDoS vulnerabilities in case of complex regexes due to the time or space limits.

Another family of ReDoS generators (e.g. `Rescue` [83], `Revealer` [60]) combine dynamic and static techniques. They use the static information about the regex in a form of an extended NFA to guide the subsequent genetic search or to simulate the matching process of a regex.

None of these detectors aims to detect vulnerable regexes for automata-based algorithms and focus on exploiting repetition for backtracking algorithms (they are indeed not efficient as we show in our experiments in Section 5.3).

1.5 Contributions of the Thesis

In this thesis, we achieved the following contributions:

The first large-scale study of vulnerability of automata-based matching. It has been recognized that regexes with bounded repetition can suffer from performance problems both in backtracking and automata-based matchers. Until now, this problem has never been studied systematically, and possibilities of exploiting it for ReDoS have not been analyzed. We evaluate a set automata-based regex matchers (`RE2`, `grep`, `Hyperscan`, `SRM`) as well as standard library backtracking matchers of `.NET`, `Python`, `Perl`, `PHP`, `Java`, etc. on a comprehensive database of regexes. We confirm that regexes with bounded repetition are potentially vulnerable to ReDoS attacks not only for backtacking matchers but also for automata-based matchers even though they are considered more robust. We further find that if a regex does not contain bounded repetition, it mostly cannot be used to perform a ReDoS attack on automata-based matchers. Thus, bounded repetition seems to be almost the only source of performance problems for the automata-based approach (not considering advanced features such as back-references).

A new method for discovering ReDoS vulnerabilities. The state-of-the-art ReDoS detectors focus only on backtracking regex matchers leaving the automata-based regex engines unnoticed. We present a methodology for generating evil texts that targets automata-based matchers and regexes with bounded repetition. It is based on the observation that automata-based regex engines generate a part of the state space of the DFA. We aim at forcing them to generate a large number of large states.

We perform an extensive experimental evaluation of our ReDoS generator against other state-of-the-art ReDoS generators, on a large set of practical regexes, with a comprehensive set of automata-based and backtracking matchers. We also discover ReDoS attack vectors on state-of-the-art real-world security applications represented by Snort using `Hyperscan` and the HW-accelerated regex matching engine on the NVIDIA BlueField-2 card. The experiments show that our generator is significantly more successful in creating ReDoS attacks on regexes than the current state-of-the-art ReDoS generators, especially on automata-based matchers and on regexes with bounded repetition.

Efficient algorithm for matching bounded repetition. The existing automata-based techniques are constructed by compiling a regex to an NFA and subsequent determinisation of the NFA into a DFA. The problem is that in the case of regexes with bounded repetition, the worst-case complexity of the DFA simulation is exponential in the length of the text and double-exponential in the size of the regex.

We aim to find a solution of efficient matching for a class of highly nondeterministic regexes with bounded repetition by using *counting automata* (CAs). First, a regex is converted into a nondeterministic CA which is then determinised. The main hurdle, however, is to find a succinct deterministic representation that can be used to implement fast matching. This has been an open problem (whose importance was stressed, e.g., in [92]) that a number of other works, such as [51, 56], have attempted to solve, but they could only cope with very restricted fragments or alleviate the problem only partially,

yielding solutions of limited practical applicability only. For example, the naive determinisation of CAs which encodes counter values as parts of control states can easily lead to a state explosion.

We make the first step towards succinct determinisation of CA using the generalized subset construction. Our algorithm can produce deterministic CAs exponentially more succinct than the corresponding DFAs. We also develop a simplified and faster version of the general algorithm for the sub-class of so-called *monadic CAs* (MCAs), i.e., CAs with counting loops on character classes, which are common in practice. Its worst-case complexity is only polynomial in the maximum values of counters.

We elaborate these ideas into a succinct transformation of CAs into deterministic *counting-set automata* (CsAs). We propose a novel compilation of regexes to CAs which generalizes the Antimirov's derivative construction. It is cheap and produces automata without ϵ-transitions whose size is independent of the repetition bounds and linear in the size of the regex. The CA is then determinised into a deterministic CsA—a deterministic automata with a special type of registers which can hold the so-called *counting sets*, i.e., a set of bounded integer values. The counting sets support a limited selection of simple set operations that can be implemented to run in constant time regardless of the size of the set. The main advantage of this workflow is that the size of the produced CsAs is independent of the repetition bounds and matching is linear in the length of the text.

We design a framework for matching based on CsA simulation that is applicable to a relatively large sub-class of regexes. The results of our experimental evaluation show that our framework handles the majority of regexes with bounded repetition found in practice. Moreover, it is much more stable and outperforms the state-of-the-art matchers regardless of the sizes of the repetition bounds.

1.6 Publications

The following lists the published papers the text of this thesis is based on.

APLAS'19. In [49], we proposed an efficient algorithm for determinising CAs into deterministic CAs. The algorithm avoids unfolding bounded repetition into control states, unlike the naive approach, and thus produces much smaller deterministic automata.

OOPSLA'20. In [98], we aimed to reduce the representation of regexes with bounded repetition even more using counting sets. The algorithm determinises the CAs into CsAs instead of deterministic CAs. The size of CsAs is independent of the repetition bounds and the matching is linear in the length of the input text.

USENIX Security'22. In [100], we studied the performance characteristics of automata-based regex matchers and their vulnerability against ReDoS attacks. We focused especially on regexes that use bounded repetition. We proposed a new ReDoS generator that can generate attacks on automata-based matchers. We achieved a slowdown of regex matching engines on regexes with bounded repetition by orders of magnitude and showed evidence that the bounded repetition is indeed a serious security issue even for automata-based matchers.

Other publications. Further, I published papers that are not directly related to the topic of this thesis.

In [68], we explored a possibility of improving existing methods for verification of parallel systems. We particularly concentrated on safety properties of well-structured transition systems. We experimented with a simple refinement algorithm based on analysing minimal runs and succeeded in generating significantly more succinct invariants than the state-of-the-art methods.

In [101], we proposed a refined version of the Parikh image abstraction of finite automata to resolve string length constraints. We integrated this abstraction into the string solver SLOTH. The experimental results showed that our extension of SLOTH has good results on simple benchmarks, as well as on complex benchmarks that are real-word combinations of transducers and concatenation constraints.

1.7 Outline of the Thesis

Chapter 2 contains preliminaries on regexes, finite automata and existing approaches to pattern matching and conversion of a regex to an automaton. Chapter 3 presents the results of [49]. In Chapter 4, we introduce the contributions from [98]. The results of [100] are discussed in Chapter 5. Chapter 6 concludes the thesis and discusses the future work.

'Mathematics takes us into the region of absolute necessity, to which not only the actual word, but every possible word, must conform.'

Bertrand Russell

2 Preliminaries

In this chapter, we define basic notions used later in the thesis, in particular regular expressions and languages, extended regular expressions and finite automata. Then we introduce the existing approaches to pattern matching and conversion of regular expressions to automata.

2.1 Kleene Regular Expressions and Languages

The notion of regular expressions was first described in 1956 by S. C. Kleene in his seminal paper [58] on finite automata theory. In his paper, he presented a theorem which states that any regular language is accepted by a finite automaton and conversely that any language accepted by a finite automaton is regular.

Let us now formally defined a notion of *regular expressions* (*regexes*). Let Σ be *alphabet* of *characters/symbols*. *Words* are sequences of symbols $a_1 \ldots a_n \in \Sigma^*$. We use ϵ to denote the *empty word*. *Languages* are sets of words. The concatenation of words u and v is denoted as $u \cdot v$ (often abbreviated to uv) and is lifted to sets as usual. We call $a \in \Sigma$ the *head* of the word $a.w$ and $w \in \Sigma^*$ its *tail*. Furthermore, we write L^n for the n-th power of $L \subseteq \Sigma^*$ with $L^0 \stackrel{\text{def}}{=} \{\epsilon\}$ and $L^{n+1} \stackrel{\text{def}}{=} L^n \cdot L$. The syntax of regexes is the following:

$$R ::= \varepsilon \mid a \mid (R) \mid R\,R \mid R|R \mid R*.$$

The *language* of a regex R, $L(R)$, is constructed inductively to the structure of R, from its atomic sub-expressions using the language operations denoted by the regex combinators. They are understood as usual:

- $L(\epsilon) \stackrel{\text{def}}{=} \epsilon$,
- $L(a) \stackrel{\text{def}}{=} \{a\}$ for $a \in \Sigma$,
- $L(R_1 R_2) \stackrel{\text{def}}{=} L(R_1) \cdot L(R_2)$,
- $L(R_1|R_2) \stackrel{\text{def}}{=} L(R_1) \cup L(R_2)$, and
- $L(R^*) \stackrel{\text{def}}{=} L(R)^*$.

R is *nullable* if $\epsilon \in L(R)$.

2.2 Extended Regular Expressions

Extended regular expressions (ERE) enrich the basic regexes with special operators. They increase the succinctness and expressive power of the basic regexes. ERE are supported by most of the programming languages, including Java, Python, Perl, or JavaScript. Even though there have been several attempts at standardisation of ERE, none has been unanimously accepted by the regex-engine developer community. The most common standards of ERE are *the Perl Compatible Regular Expression* (PCRE) [47] and *the POSIX Extended Regular Expressions* [5]. They have common core semantics but differ in extended features. The most of the programming languages support PCRE syntax and semantics. It is used by many open source projects including `PHP`, `Apache`, `R`, `Postfix`, `KDE` etc. We will now describe the main extensions defined by PCRE:

- **Bounded repetition (or bounded quantifier/counting operator)**: *Bounded repetition* is the main focus of this thesis. A bounded repetition of a regex R is denoted as:

  $$R\texttt{\{n,m\}}$$

  where $n, m \in \mathbb{N}$, $0 \leq n$, $0 < m$, $n \leq m$. It specifies concatenations of n to m strings that conform to R, formally:

  $$L(R\{n,m\}) \stackrel{\text{def}}{=} \bigcup_{i=n}^{m} (L(R))^i.$$

  For example '`(Help!){2,10}`' represents all words where '*Help!*' appears 2–10 times.

  In this thesis, by *regex* we will usually mean the basic syntax of regexes extended with the bounded repetition and *custom* and *build-in character classes* discussed below.

- **Custom character classes**: *Custom character classes* represent sets of characters. Character classes are most often of the form '`[a-z]`' denoting an interval of characters, '`[^a-z]`' denoting a complement of the interval of characters, and '`a`' denoting a singleton containing the character $a \in \Sigma$. The intervals can be subtracted from each other and are in general closed under Boolean operations.

- **Build-in character classes**: *Build-in character classes* are standard designation of selected, frequently used sets of characters. They are represented by escaped characters. For example, '`\d`' represents a decimal digit, or '`\s`' represents a white space character. A special and most used build-in character class is the class of all characters, denoted by '`.`'.

- **Capturing groups**: Subexpressions closed in parenthesis are called *capturing groups*. They are relevant to a special mode of matching when the matching engine returns an array of parts of the match where each item corresponds to one of the capturing groups. The groups can be numbered and later reference with *backreferences* (discussed below).

- **Non-capturing groups**: *Non/capturing groups* are subexpressions of the form '`(?:exp)`'. The subexpression functions as a single unit but does not save the matched string in the results array.

  For example, a regex '`(?:Marry|John)\s(.*?)\s(?:Jones|Williams)`' matches the fist name, '`(?:Marry|John)`', the second name, '`(.*?)`', and the family name, '`(?:Jones|Williams)`', without saving the first or family names in the results array.

- **Back-references**: *Back-references* are regexes commands which refer to the previous matched subexpressions designated with capturing groups. They are represented by a backslash followed by a digit greater than 0.

  For example, '`<([A-Z])>.*?</\1>`' would match a pair of opening and closing HTML tags with the text in between, i.e, `<B>bold text</B>`.

  It is known that the matching problem of regexes with backreferences is NP-complete [3], and decision problems (e.g. equivalence, universality or inclusion) are undecidable [37]. Back-references extend the power of regexes beyond regularity.

- **Greedy and lazy quantifiers**: The operators `?`, `+`, `*`, `{n}`, or `{n, m}` are called *quantifiers*. By default, the quantifiers are interpreted as *greedy*. If a greedy quantifier is used, then the quantified subexpression is matched as many times as possible. On the other hand, if a *lazy* quantifier is used, then the quantified subexpression is matched the minimal number of times.

  For example the greedy quantifier in the regex '`".+"`' used against the input text

  "Begin at the beginning", the King said gravely, " and go on till you come to the end: then stop."

  would return the whole input text. The lazy quantifier in a regex '`".+?"`' would match only *"Begin at the beginning"*.

- **Curet and dollar metacharacters**: The metacharacter '`^`' represents the beginning of a string/line. If there is no '`^`' at the beginning of the regex then the match can appear anywhere in the input string. The metacharacter '`$`' represents the end of a string/line. Regexes with '`^`' at the beginning or '`$`' at the end are said to be *anchored* patterns.

Character classes and different character encoding standards. In practice, different character encodings are used. Different encodings influence how different regexes are interpreted. There are two main character encodings *Unicode* and *ASCII*.

American Standard Code for Information Interchange (ASCII) was first used in 1981[1] by the International Business Machines Corporation (IBM) to facilitate the representation, storing, and interchange of data among computers and other devices [18]. Every bit of the 7-bit binary vector represents a single character. ASCII is able to encode a set of 128 characters–95 graphic symbols, such as (upper and lower case of) alphabet, digits, math signs, punctuation marks and 33 control symbols, such as '`\n`','`\r`',

[1] When my supervisor was born.

'`\t`', etc. [43]. It was widely used until December 2007, when UTF-8 has become the most used encoding [28].

Unicode is a universal character encoding standard that assigns to each character a unique code (a *code point*) [105]. Unicode can potentially produce over 1 million code points, however, the Unicode standard now encompasses 144,076 characters. While ASCII can represent only the letters, digits, and punctuation, Unicode supports all characters used in almost every language in the world including favorite emojis [53]. Unicode can be implemented by different character encodings.

The most widely used encoding is the 8-bit Unicode Transformation Format UTF-8—used by 97.5 % of all the websites whose character encoding is known [104]. UTF-8 is a byte oriented format with a variable length. It represents a symbol by a 1–4 byte code. For example, a symbol '`@`' is represented by a code '`\x40`'. UTF-8 is backward compatible with ASCII.

The difference between these two encodings is most evident in the interpretation of build-in character classes. In our experiments, it makes comparison of the matching engines difficult. For example, the expression '`\w`' represents in ACSII a character class '`[a-zA-Z0-9_]`'. In Unicode, '`\w`' represents a wider character class which is not precisely specified. It includes a character class '`[a-zA-Z0-9]`', connector punctuation (like the underscore), diacritics, such as '˜', most characters that can be part of a word in any language (Thai letters, Greek letters, etc.), and the modifier letters, which are generally used to add auxiliary markings to letters [1, 36]. Hence, '`\w{8}`' would match with '*čipmánek*' in UNICODE encoding while in ASCII encoding not.

2.3 Finite Automata

We consider *nondeterministic finite automata (NFA)* over the alphabet Σ of the form $A = (Q, \delta, q_0, F)$ where

- Q is a finite set of *states*,
- δ is a set of *transitions* of the form $q \text{-}\{a\}\!\!\rightarrow r$ with $q, r \in Q$,
- $a \in \Sigma$, $q_0 \in Q$ is the *initial state*, and
- $F \subseteq Q$ is the set of *final states*.

The *language* of the automaton, denoted $L(A)$, is the set of all words $a_1 \dots a_n$, $n \geq 1$, for which the automaton has an *accepting run*, a sequence of transitions $q_0 \text{-}\{a_1\}\!\!\rightarrow q_1 \text{-}\{a_2\}\!\!\rightarrow \cdots \text{-}\{a_n\}\!\!\rightarrow q_n$ with $q_n \in F$. The empty sequence is a run with $q_0 = q_n$ over ϵ.

The automaton is *deterministic (DFA)* if for every state q and symbol a, δ has at most one transition $q \text{-}\{a\}\!\!\rightarrow r$. Any NFA can be determinised by the *subset construction*, which creates the DFA $A' = (Q', \delta', q'_0, F')$ with

- $Q' = 2^Q$, i.e., with subsets of A as the new states,
- the singleton $\{q_0\}$ as the initial state q'_0,
- with sets intersecting with F being final, i.e., $F' = \{S \subseteq Q \mid S \cap F \neq \emptyset\}$, and

- with the successor of a state $S \subseteq Q$ under a symbol a constructed as the set of a-successors of the NFA states in S, i.e. $S \text{-}\{a\}\!\!\rightarrow S' \in \delta'$ for $S' = \{s' \mid s \in S \wedge s \text{-}\{a\}\!\!\rightarrow s' \in \delta\}$.

2.4 Pattern Matching

Pattern matching is the process of checking whether an input string/or its part belongs to the language described by a regex.

- **Partial match.** If only a substring of the input string is required to be in the language of the regex, then we refer to it as a *partial match*. In other words, it is checked whether the input string w can be written as a concatenation $x.v.y$ such that $v \in L(R)$, i.e., $w \in L(\texttt{.*}\ R\ \texttt{.*})$.
- **Exact match.** If the entire input string is required to be in the language of the regex, then we refer to it as an *exact match*. In other words, it is checked whether the input string w is in $L(R)$.

As already mentioned in Section 2.2, some extended regexes consist of *anchors*–'`^`' at the start of the regex and '`$`' at the end of the regex. The anchors are interpreted in different matching modes (*single* or *multi-line mode*) differently. Let an input string consist of several lines. In a single-line mode, '`^`' is considered the beginning of the input string and '`$`' the end. In a multi-line mode, '`^`' represents the beginning of the line and '`$`' the end. The anchors are used to find an exact match in a single-line mode.

Example 2.4.1. Let us have an input string '*Today is Saturday!\nTommorow is Sunday!*' and the regex '`^To.*day!$`'. In the multi-line mode, '`^`' matches before the first and second occurrence of *T*. '`$`' matches between the first '*!*' and '\n', and also after the second '*!*'. In the single-line mode, '`^`' matches only before the first '*T*', and '`$`' matches only after the second '*!*'. □

Next, we will provide an overview of approaches to pattern matching. The approaches are classified into *backtracking* and *automata-based*.

Backtracking. Backtracking [86] in its simplest form is a recursive procedure that descends the syntactic tree of the regex while reading the text from the left to the right and matching its characters against subexpressions of the regex. Since disjunction and iteration offer a choice, the recursion tries the longest path through the NFA and if the matching fails, it backtracks to the last unexplored choice. It is in fact very similar to a depth-first exploration of all runs following the input line through an NFA corresponding to the regex.

Example 2.4.2. Let us have a regex '`unit|unicode|unicorn`' and an input string '*It is a unicorn!*' (see Figure 2.1). First, the matcher tries to match '*I*' from the input string and '`u`' from the regex. It fails to match, as well as the 2nd letter '*t*' of the input string. The matcher unsuccessfully continues in attempts to match until it matches the 10th letter of the input string '*u*'. At this point, there are three alternative paths through NFA–matching '`unit`', '`unicode`' or '`unicorn`'.

input text	*pattern*	*match*	*input text*	*pattern*	*match*
Saturday	`day`	*yes*	*days*	`day`	*yes*
Saturday	`^day`	*no*	*days*	`^day`	*yes*
Saturday	`day$`	*yes*	*days*	`day$`	*no*
Saturday	`^day$`	*no*	*days*	`^day$`	*no*

Table 2.1: Having an input string '*Saturday*', '`^day`' and '`^day$`' do not match it since '`day`' does not match at the beginning of the input string. Having an input string '*days*', the last two options '`day$`' and '`^day$`' do not match it since '`day`' does not match at the end of the input string.

A backtracking matcher usually processes the alternatives from left to right. First, it tries to match '`unit`', concretely '`u`' from the regex and '*u*' from the input string. Since these letters match, it continues. Also '*n*' and '*i*' from the input string match. The next *c* does not match and since there were other alternatives, the matcher backtracks to the 9th letter of the input string and tries to match '`unicode`'. It matches '*u*', '*n*', '*i*', '*c*', and '*o*', then it fails while matching '*r*'. The matcher backtracks to the same point as in the previous case and tries the last alternative '`unicorn`' when it finds a match. □

Backtracking algorithms are probably the most often implemented ones, e.g., in the standard libraries of `.NET`, `Python`, `Perl`, `PHP`, `Java`, and `JavaScript`. Backtracking matchers are conceptually very simple (a basic implementation takes a few lines of functional code, e.g. [76]) and they are processing a single path through the NFA at a time. Thus, they are flexible and amenable to easy extensions with features such as priority of matched sub-expressions, submatching, or back-references.

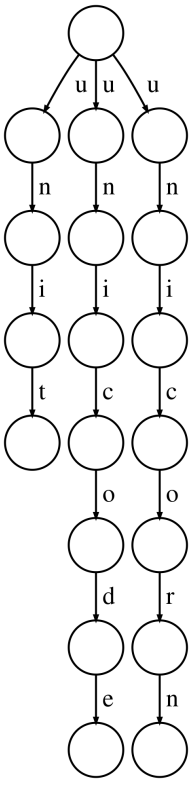
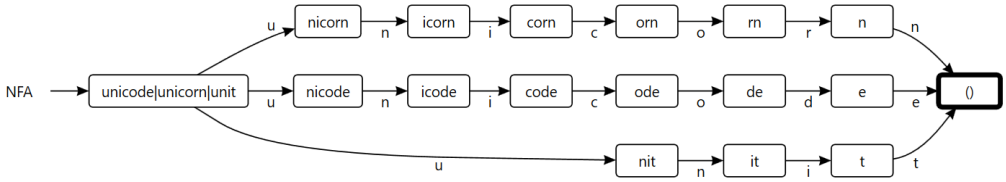
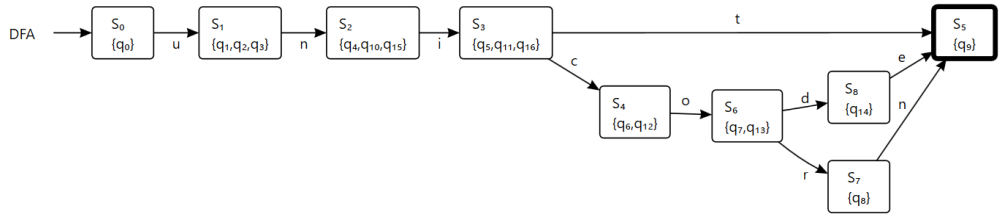


Figure 2.1: Example of backtracking approach.

Backtracking needs almost no preprocessing, since the preprocessing phase involves only compilation of a regex. Nonetheless, as the number of NFA runs over a single line is in the worst case exponential in its length, the worst-case complexity of matching using a backtracking matcher is *exponential in the length of the text*. Hence, backtracking-based algorithms may easily require a prohibitively large time, and are quite prone to ReDoS attacks, cf. [72]. Extreme matching times do not occur often if regexes are written defensively, and modern implementations are fast, especially when an accepting path is guessed early. However, overlooking a dangerous regex is easy and writing such a regex intentionally is even more so. For



(a) NFA for the regex '`(unit|unicode|unicorn)`'



(b) DFA for the regex '`(unit|unicode|unicorn)`'

Figure 2.2: NFA for regex '`^(unit|unicode|unicorn)$`' has 17 states. A corresponding DFA is created using a subset construction and has only 9 states.

instance, when run on the regex '`(a|b|ab)*bc`' against the input string '$(ab)^n ac$' with $n = 50$, standard matchers in `Java`, `Python`, `.NET` become unresponsive [76].

Offline DFA-similation. A basic automata-based alternative to backtracking is *offline DFA-simulation*. Offline DFA-simulation is based on construction of the DFA up front in the preprocessing phase. Having the DFA at hand is the best scenario for matching since every character is then processed in constant time by simply following the unique transition from the current DFA state to the successor. Figure 2.2 illustrates the difference between NFA and DFA for a regex '`^(unit|unicode| unicorn)$`'. An issue with determinisation is that it may explode exponentially, rendering matching slow or unfeasible (the matcher may time out already during the DFA construction). Moreover, it is hard to extent them with features such as back-references. This approach is therefore seldom used in practice.

NFA-simulation. A viable alternative to the offline DFA-simulation is based on *Thompson's algorithm* [86] aka the *NFA-simulation*, which determinises only the part of the given NFA used to match the given word. Hence, the preprocessing cost is low. The NFA-simulation essentially differs from the backtracking algorithm by replacing the depth-first NFA exploration strategy by a breadth-first search strategy. Reading each symbol a of the input string means updating the set of all NFA states reached by runs over the so-far processed prefix of the string. The time needed to process a single symbol is thus linear to the size of the NFA (an iteration through all a-transitions starting in the current set), and the entire matching is only linear in the length of the string.

An advanced implementation of the NFA simulation is a part of Intel's `Hyperscan` [20] (among a number of other techniques such as advanced use of the Boyer-Moore algorithm [14] for string-matching, innovative parallelisation, or specialised processor instructions).

However, the overhead during matching using NFA-simulation might be significant, and the construction can still explode on some strings.

Online DFA-simulation. Online DFA-simulation is based on the NFA-simulation. A crucial optimization on top of NFA-simulation is caching. The reached sets of NFA states are actually states of the DFA constructed by the subset construction, while a DFA state and its successor reached after reading a symbol constitute a DFA transition. The encountered DFA states and transitions are cached. The management of the cache may be the following (as implemented in `RE2` [42]): (i) When the cache exceeds some size, it is reset and (ii) if the cache utilization is too low or is reset too often, the matcher disables the cache completely and reverts to pure NFA-simulation. The online DFA-simulation may run into trouble from two reasons:

1. **Many states.** If many new states are discovered, the matcher cannot use the precomputed states from the cache and many cache misses occur.

2. **Large states.** It is time consuming to compute large states.

When the matching algorithm stays inside the cache, it is exactly the same as the offline DFA-simulation, with constant per-character complexity. Hence, the online DFA-simulation can achieve much better performance and especially stability and resilience against ReDoS than NFA-simulation. The disadvantage is perhaps a less straightforward implementation, which implies lower flexibility. It is not clear how to extend online DFA-simulation with advanced regex features such as back-references. Well-known examples of industrial matchers based on the DFA-simulation include `RE2` [42], `grep` [32], `SRM` [82], or the regex matcher in `Rust` [31].

2.5 Conversion of Regular Expressions to Automata

Several methods of converting a regex to a corresponding automaton have been proposed. The basic methods are McNaughton and Yamada's, Thompson's algorithm, Glushkov's algorithm, or algorithms based on Brzozowski or Antimirov's derivatives [64, 86, 40, 13, 13, 4]. Many papers were published proposing their optimization. In the following paragraph, we will present only a brief overview of these algorithms and then we will focus on the constructions based on derivatives which we build on in this thesis.

The Thompson's algorithm converts a given regex into an abstract syntax tree and then builds a corresponding NFA by traversing the tree bottom-up from the leaf nodes to the root of the tree. The resulting NFA has $\Theta(n)$ states where n is the length of the regex. The McNaughton and Yamada's NFA is constructed from a linearized version of a given regex, that is, one in which the symbols are distinguished according to their position in the regex. The number of states of the NFA is $\Theta(s)$ where s is the number of occurrences of the symbols in the regex. The Glushkov's construction produces NFA with $\Theta(n)$ states. The NFA is also constructed from a linearized version of a given regex. First, it inductively computes three sets of symbols—First, Last and Follow: the sets of the first letters, last letters, and letters that follow the first letters for the linearized regex. Then it constructs the equivalent NFA directly from these sets. The output of the Glushkov's algorithm is similar to the Thompson's construction, once the ϵ-transitions are removed.

Regexes and their derivatives. In 1964, Brzozowski introduced word *derivatives* of regexes [13] and suggested an algorithm which converts a regex into a DFA where the word derivatives serve as states of the DFA. His approach easily supports also regexes extended with Boolean operations such as complement or conjuction. The notion of derivative was generalized to partial derivatives by Antimirov [4]. The Antimirov's construction yields a succinct NFA of the size linear to the number of states and quadratic to the number of transitions. It was proven by [16] that the Antimirov's NFA is the quotient of the Glushkov automaton for some equivalence relation. Antimirov's derivatives have been recently extended to unrestricted regexes (regexes extended to complementation and intersection) [15]. The difference between Antimirov's and Brzozowski approaches can be seen in Figure 2.3.

Definition 2.1. *Let Σ be an alphabet. The derivative of a language $L \subseteq \Sigma^*$ with respect to a string $u \in \Sigma^*$ is defined in [73] to be:*

$$\partial_u(L) = \{v \mid uv \in L\}.$$

In other words, the derivative of L with respect to u is the set of strings v whose concatenation with u as the prefix belongs to L. For example, let us have a regex $R =$ '`(unit|unicode|unicorn)`' with a language $L(R) = \{unit, unicode, unicorn\}$. Derivatives of $L(R)$ with respect to a string '`u`' is a set of strings $\partial_u(L(R)) = \{nit, nicode, nicorn\}$.

Definition 2.2. *Let Σ be an alphabet, $L \subseteq \Sigma^*$ is a language and R, S are regexes over Σ, i.e., $L(R), L(S) \subseteq \Sigma^*$. We say that R and S* are equivalent, *written $R \equiv S$, if $L(R) = L(S)$. We write $[R]_\equiv$ for the set $\{S \mid R \equiv S\}$, the equivalence class of R under $\equiv$.*

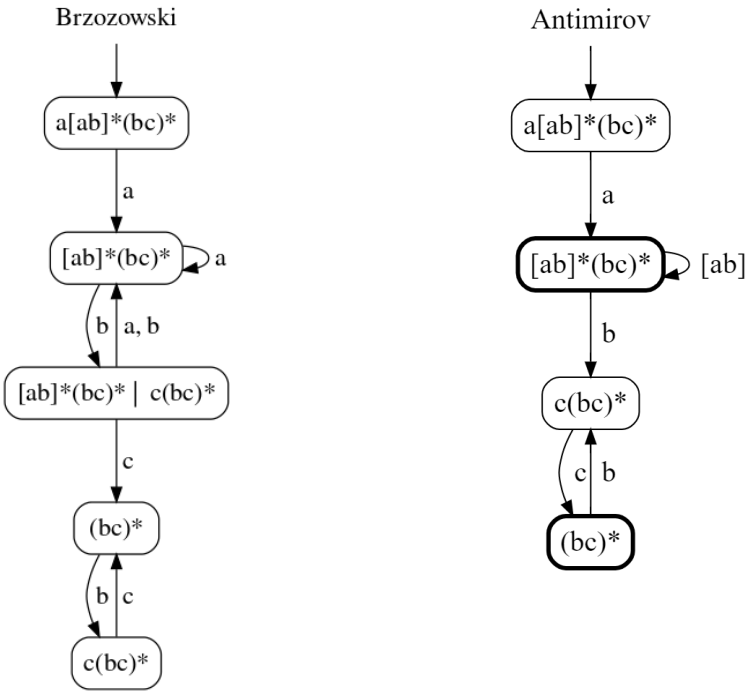
Brzozowski derivatives provide an elegant approach for converting a regex directly into a DFA. Efficient determinisation based on Brzozowski derivatives was first investigated in [8]. They can be used not only for efficient matching [73, 34] but also match generation [82]. An on-the-fly Brzozowski derivative construction is implemented for example in `SRM` [82].

The relation between the derivatives of a language and Brzozowski derivatives of a regex is that for each regex R and a symbol a it holds that $\partial_a(L(R)) = L(\partial_a^B(R))$. Let Σ be an alphabet, $L \subseteq \Sigma^*$ is a language, $a, b \in \Sigma$ and R, S are regexes over Σ. The Brzozowski derivatives of regex with respect to a symbol a are defined as follows [73]:

- $\partial_a^B(\epsilon) = \emptyset$,
- $\partial_a^B(a) = \epsilon$,
- $\partial_a^B(b) = \begin{cases} \emptyset & \text{for } b \neq a, \\ a & \text{for } b = a, \end{cases}$
- $\partial_a^B(\emptyset) = \emptyset$,

and derivatives of regexes that use operators are:

- if $\epsilon \notin L(R)$: $\partial_a^B(R \cdot S) = \partial_a^B(R) \cdot S$,
- if $\epsilon \in L(R)$: $\partial_a^B(R \cdot S) = \partial_a^B(R) \cdot S \mid \partial_a^B(S)$,



(a) DFA constructed using Brzozowski derivatives

(b) NFA constructed using Antimirov's derivatives

Figure 2.3: Construction of FA for a regex '`a[ab]*(bc)*`' using Brzozowski and Antimirov's derivatives. The result of the Brzozowski derivatives (Figure 2.3a) is a DFA. The result of Antimirov's derivatives (Figure 2.3b) is an NFA.

- $\partial_a^B(R^*) = \partial_a^B(R) \cdot R^*$,
- $\partial_a^B(R \mid S) = \partial_a^B(R) \mid \partial_a^B(S)$.

Given a regex R the Brzozowski derivatives induce a DFA $A = (Q, \Sigma, \delta, q_0, F)$. States Q include $[R]_\equiv$ and the equivalence classes of all regexes that arise in derivatives constructed from R by repeated derivation wrt Σ. The initial state is $q_0 = [R]_\equiv$, the transition function is defined by $\delta([S]_\equiv, x) = [\partial_x^B(S)]_\equiv$ for all $x \in \Sigma$, $[S]_\equiv \in Q$, and the set of final states is $F = \{[S]_\equiv \in Q \mid \epsilon \in L(S)\}$. Then A recognizes the language $L(R)$. Brzozowski proved that there are finely many derivative classes of a regex, which guarantees the termination of the DFA construction.

Antimirov's derivatives are a generalisation of Brzozowski derivatives to NFAs. In the classical setting, Antimirov derivatives are used to construct NFAs from regexes, and may in some cases result in exponentially more succinct automata than the corresponding DFAs constructed with Brzozowski derivatives. The Antimirov's construction has also been generalized to regexes extended to complementation and intersection [15, 13].

The relation between the derivatives of a language and Antimirov's derivatives of a regex is that for each a regex R it holds that $\bigcup_{R_i \in \partial_a^A(R)} L(R_i) = \partial_a(L(R))$. The Antimirov's derivatives of a regex with respect to a symbol are defined as follows [13]:

- $\partial_a^A(a) = \{\epsilon\}$,
- $\partial_a^A(b) = \begin{cases} \emptyset & \text{for } b \neq a, \\ \{\epsilon\} & \text{for } b = a, \end{cases}$
- $\partial_a^A(\epsilon) = \emptyset$,
- $\partial_a^A(R^*) = \{R' \cdot R^* \mid R' \in \partial_a^A(R)\}$,
- $\partial_a^A(R \mid S) = \partial_a^A(R) \cup \partial_a^A(S)$,
- if $\epsilon \in L(R)$: $\partial_a^A(R \cdot S) = \{T \cdot S \mid T \in \partial_a^A(R)\} \cup \partial_a^A(S)$,
- if $\epsilon \notin L(R)$: $\partial_a^A(R \cdot S) = \{T \cdot S \mid T \in \partial_a^A(R)\}$.

Given a regex R the Antimirov's derivatives induce an NFA $A = (Q, \Sigma, \delta, q_0, F)$. States Q include $[R]_\equiv$ and the equivalence classes of all regexes that arise in derivatives constructed from R by repeated derivation wrt Σ. Given a state represented by a regex $[R]_\equiv$, for each $a \in \Sigma$ and each partial derivative $R_i \in \partial_a([R]_\equiv)$, there is a transition $[R]_\equiv \text{-}\{a\}\!\!\rightarrow [R_i]_\equiv$ of A. A has the initial state $q_0 = [R]_\equiv$ and the set of final states $F = \{[S]_\equiv \in Q \mid \epsilon \in L(S)\}$.

'Pure mathematics is, in its way, the poetry of logical ideas.'
Albert Einstein

3

Succinct Determinisation of Counting Automata via Sphere Construction

As we already mentioned in the introduction, matching of regexes with bounded repetition is a challenge for all existing regex matchers. The worst-case complexity of matching based on DFA-simulation is linear in the length of the text, but double-exponential in the size of the regex—exponential to the repetition bounds. Therefore, we aim at designing alternative representation less dependent on the repetition bounds that still allows matching in time linear in the length of the text.

We propose a solution based on representing bounded repetition in automata symbolically via counters. Several such finite automata counterparts of counting constraints have appeared in the literature (e.g. [51, 56, 89, 92]), all essentially boiling down to variations on counter automata with counters limited to a bounded range of values. In this thesis, we will define our variant that we call *counting automata* (CAs). First, a regex is converted into a nondeterministic CA with $O(|R|)$ states where $|R|$ is the size of the regex. The main hurdle, however, is to convert the CA to a succinct deterministic machine that could be simulated fast in matching. The naive determinisation of CAs using the standard subset construction wastes the succinctness of counters and can easily lead to state explosion causing the approach to fail. For example, the NFA from Example 1.3.1 will have at least k states with k being the upper bound of the counter, which is exponential in the regex size, and the minimal DFA will have 2^{k+1} states. A more sophisticated method of determinisation is needed.

In this section, we present our first steps towards this end which we published first in [49]. We propose an algorithm that can produce deterministic CAs exponentially more succinct than the corresponding DFAs. We also propose a version of the algorithm restricted to repetition of *character classes*, called *monadic counting* here (e.g., '`[ab]{10}`' is monadic while '`(ab){10}`' is not). The worst-case complexity of this *specialised* algorithm is only polynomial in the maximum counter bounds (in contrast to the exponential naive construction).

We have implemented the monadic CA determinisation and evaluated it on real-life datasets of regexes with monadic counting. The experiments confirmed that our resulting deterministic CAs can be much smaller than minimal DFAs, are less prone to explode, and that our algorithm, though not optimised, is overall faster than the naive determinisation that unfolds counters. We also confirmed that monadic regexes present an important subproblem, with over 95 % of regexes in the explored datasets being of this type.

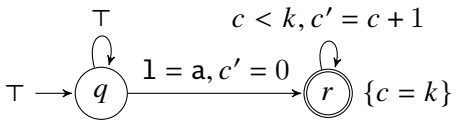


Figure 3.1: A CA for the regex '`.*a.{k}`' with $k \in \mathbb{N}$, $I : \mathtt{s} = q$, $F : \mathtt{s} = r \wedge c = k$, and $\Delta : q \text{-}\{\top,\top\}\!\!\rightarrow q \vee q \text{-}\{\mathtt{l}=\mathtt{a},c'=0\}\!\!\rightarrow r \vee r \text{-}\{c<k,c'=c+1\}\!\!\rightarrow r$.

Running example. To illustrate our algorithms, consider the regex '`.*a.{k}`', $k \in \mathbb{N}$ from Example 1.3.1. The regex corresponds to the nondeterministic CA of Figure 3.1. In the transition labels, the predicates over the variable `l` constrain the input symbol, the predicates over c constrain the current value of the counter c, and the primed variant of c, i.e., c', stands for the value of c after taking the transition. The initial value of c is unrestricted, and the automaton accepts in the state r if the value of c equals k. Our monadic determinisation algorithm, presented in Section 3.4, then outputs the deterministic CA (DCA) of Figure 3.3 (for $k = 1$). Intuitively, it uses $k + 1$ counters to remember how far back the last $k + 1$ occurrences of a appeared. Depending on k, the resulting DCA has $k + 2$ states, $4(k + 1) + 1$ transitions, and $k + 1$ counters. That is, its size is linear to k in contrast to the factor 2^k in the size of the minimal DFA (Figure 3.2). □

Outline. In Section 3.1, we introduce the state of the art on automata and regexes with bounded repetition. Section 3.2 contains preliminaries on labelled transitions systems and counting automata. In Section 3.3, we introduce an algorithm for determinising counting-automata. Section 3.4 presents a simplified and more efficient version of the determinisation algorithm that targets regexes with counting limited to character classes. In Section 3.5, we present the results of our experimentation where we compare the proposed approach with the naive determinisation. Section 3.6 concludes the results of this chapter.

3.1 Related Work

In this section, we will introduce the state of the art on automata and regexes with bounded repetition, which is relevant also for Chapter 4.

Automata with counting. The use of counters has been investigated in [10] for regexes with bounded repetition, building on the formalism of counter automata called CNFAs [38]. CAs in this thesis are essentially a symbolic generalization of CNFAs with some small technical differences, such as counters being 0-based as opposed to 1-based in CNFAs. The latter difference is mainly due to our use of a generalized *Antimirov's* construction of CAs, discussed in Chapter 4, as opposed to a generalized *Glushkov* construction used in [38], which is algorithmically quite different. The work [10] focuses mostly on deterministic regexes and on a different problem than pattern matching as we defined it, namely, the so-called incremental matching in the context of database queries (a query is repeatedly evaluated on a gradually changing word). For standard matching, it uses a variant of the NFA-simulation applied directly on a CA instead of an NFA

(hence the translation of the regex to an automaton does not depend on the counter bounds, but each character of the text is processed with the same cost as with the original NFA-simulation, at worst linear to the size of the NFA and the counter bounds). This algorithm is indeed fast on deterministic regexes from practice but can slow down significantly on nondeterministic ones (which we witnessed in several experiments with the prototype implementation of [10] on several of our regexes).

The work in [55] is a theoretical study of matching regexes with bounded repetition. It proposes a matching algorithm based on dynamic programming that runs in time at worst quadratic in the length of the text (while determinisation and NFA-simulation-based algorithms run in time linear in the text length). The experimental comparison of [10] with their variant of NFA-simulation suggests that the matching algorithm of [55] is indeed not competitive in practice.

Extended FAs (XFAs) augment classical automata with a scratch memory of bits [88, 89] that can represent counters. Regexes are compiled into deterministic XFAs by first using an extended version of the NFA-simulation, followed by an extended version of the classical powerset construction and minimization. Although a small XFA may exist, the determinisation algorithm incurs an intermediate exponential blowup of the search space for inputs such as '`.*a.{k}`'.

R-automata [2] are also related to our CAs, but their counters need not have upper bounds and cannot be tested or compared. Further, there are various notions of extended finite state machines whose expressive power goes beyond regular languages, e.g., [59, 84, 89, 7]. Such automata are, however, not suitable for the problem of pattern matching considered in this thesis.

Regexes with bounded repetition. Regexes with bounded repetition are also discussed in [56]. The automata with counters used in [51], called FACs, are close to our CAs, but we allow symbolic character predicates and more kinds of counter updates. The conversion from regexes to FACs proposed in [51] uses a variant of Glushkov automata [40] and the `first-last-follow` construction [9, 12].

There are also works on regexes with bounded repetition that translate deterministic regexes to CAs and work with different notions of determinism [51, 38, 17]. A central result in [51] is that *counter-1-unambiguous* regexes can be compiled into deterministic FACs and that checking determinism of FACs can be done in polynomial time. The related work in [52] studies membership in regexes with bounded repetition. None of these papers addresses the problem of determinising nondeterministic CAs.

3.2 Preliminaries

We use $\mathbb{N}$ to denote the set of natural numbers $\{0, 1, 2, \ldots\}$. Given a function $f : A \to B$, we refer to the elements of f using $a \mapsto b$ (when $f(a) = b$). Given a set of variables V and a set of constants Q (disjoint with $\mathbb{N}$), we define a *Q-formula over V* to be a quantifier-free formula φ of Presburger arithmetic extended with constants from Q and Σ, i.e., a Boolean combination of (in-)equalities $t_1 = t_2$ or $t_1 \leq t_2$ where t_1 and t_2 are constructed using $+$, $\mathbb{N}$, and V, and predicates of the form $x = a$ or $x = q$ for $x \in V$, $a \in \Sigma$, and $q \in Q$. An assignment M to free variables of φ is a *model* of φ, denoted as $M \models \varphi$, if it makes φ true.

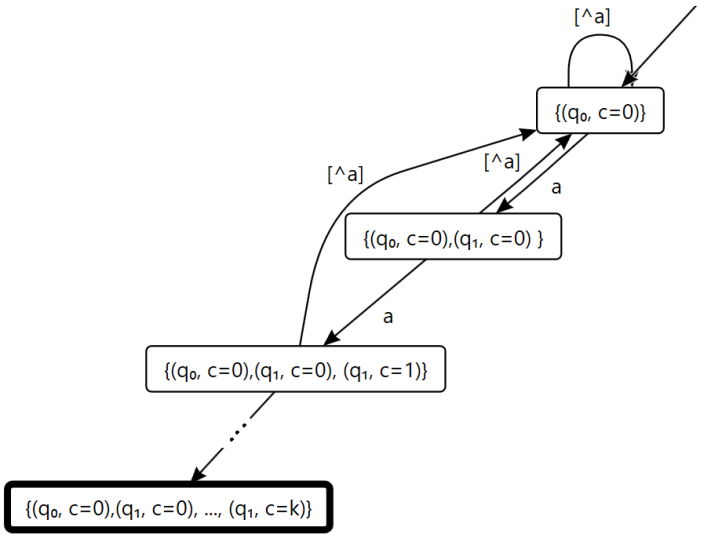


Figure 3.2: DFA simulation using subset construction constructs for a regex '`.*a.{k}`' a DFA with 2^{k+1} states. It unwinds the counter values to separate states.

Given a formula φ and a (partial) map $\theta : terms(\varphi) \to S$, where $terms(\varphi)$ denotes the set of terms in φ and S is some set of terms, $\varphi[\theta]$ denotes a *term substitution*, i.e., the formula φ with all occurrences of every term $t \in dom(\theta)$ replaced by $\theta(t)$. As usual, replacing a larger term takes priority over replacing its subterms (we treat primed variables and parameters as atomic terms, hence $(p' = 1)[\{p \mapsto q\}]$ is still $p' = 1$). The *substitution formula* φ_θ of θ is defined as the conjunction of equalities $\varphi_\theta \stackrel{\text{def}}{=} \bigwedge_{t \in dom(\theta)} (\theta(t) = t)$.

Labelled transition systems. We will introduce our counting automata as a specialisation of the more general model of labelled transition systems. This perspective and related notation allows for a more abstract and concise formulation of our algorithms than the more standard approach, in which one would define counting automata in a more straightforward manner as an extension of the classical finite automata. A *labelled transition system* (LTS) over Σ is a tuple $T = (Q, V, I, F, \Delta)$ where:

- Q is a finite set of *control states*,
- V is a finite set of *configuration variables*,
- I is the *initial Q-formula* over V,
- F is the *final Q-formula* over V, and
- Δ is the *transition Q-formula* over $V \cup V' \cup \{\mathtt{1}\}$ with $V' = \{x' \mid x \in V\}$, $V \cap V' = \emptyset$, and $\mathtt{1} \notin V$.

We call $\mathtt{1}$ the *letter/symbol variable* and allow it as the only term that can occur within

a predicate $\mathtt{1} = a$ for $a \in \Sigma$, called an *atomic symbol guard*.[1] Moreover, $\mathtt{1}$ is also not allowed to occur in any other predicates in Δ.

A *configuration* is an assignment $\alpha : V \to \mathbb{N} \cup Q$ that maps every configuration variable to a number from $\mathbb{N}$ or a state from Q.

Let C be the set of all configurations. The transition formula Δ encodes the transition relation $[\![\Delta]\!] \subseteq C \times \Sigma \times C$ such that $(\alpha, a, \alpha') \in [\![\Delta]\!]$ iff $\alpha \cup \{x' \mapsto k \mid \alpha'(x) = k\} \cup \{\mathtt{1} \mapsto a\} \models \Delta$. We use $|\Delta|$ to denote the size of $[\![\Delta]\!]$. For a word $w \in \Sigma^*$, we define inductively that a configuration α' is a *w-successor* of α, written $\alpha \xrightarrow{w} \alpha'$, such that $\alpha \xrightarrow{\epsilon} \alpha$ for all $\alpha \in C$, and $\alpha \xrightarrow{av} \alpha'$ iff $\alpha \xrightarrow{a} \bar{\alpha} \xrightarrow{v} \alpha'$ for some $\bar{\alpha} \in C$, $a \in \Sigma$, and $v \in \Sigma^*$. A configuration α is *initial* or *final* if $\alpha \models I$ or $\alpha \models F$, respectively. The *language* $L(T)$ of T is the set of all words that T accepts.

Outcome. The *outcome* of T on a word w is the set $out_T(w)$ of all w-successors of the initial configurations, and w is accepted by T if $out_T(w)$ contains a final configuration.

Counting variable. A *counting variable* (counter) is a configuration variable c whose value ranges over $\mathbb{N}$ and which can appear (within Δ, I, and F) only in *atomic counter guards* of the form $c \leq k, c \geq k$, (using $<, =, >$ as syntactic sugar) or *term equality tests* $t_1 = t_2$, and in *atomic counter assignments* $c' = t$ with t, t_1, t_2 being *arithmetic terms* of the form $d + k$ or k with $k \in \mathbb{N}$ and d being a counter. A *control state variable* is a variable $\mathtt{s}$ whose value ranges over states Q and appears only in *atomic state guards* $\mathtt{s} = q$ and *atomic state assignments* $\mathtt{s}' = q$ for $q \in Q$. A Boolean combination of atomic guards (counter, state, or symbol) is a *guard formula* and a Boolean combination of atomic assignments is an *assignment formula*.

Nondeterministic counting automaton. A *(nondeterministic) counting automaton* (CA) is a tuple $A = (Q, C, I, F, \Delta)$ such that (Q, V, I, F, Δ) is an LTS with the following properties:

1. The set of configuration variables $V = C \cup \{\mathtt{s}\}$ consists of a set of counters C and a single control state variable $\mathtt{s}$ s.t. $\mathtt{s} \notin C$.

2. The transition formula Δ is a disjunction of *transitions*, which are conjunctions of the form $\mathtt{s} = q \wedge g \wedge f \wedge \mathtt{s}' = r$, denoted by $q \text{-}\{g,f\}\!\!\rightarrow r$, where $q, r \in Q$, g is the transition's *guard formula* over $V \cup \{\mathtt{1}\}$, and f is the transition's *counter assignment formula*, a conjunction of atomic assignments to counters, in which every counter is assigned at most once.

3. There is a constant $\boldsymbol{max}_A \in \mathbb{N}$ such that no counter can ever grow above that value, i.e., $\forall c \in C\ \forall w \in \Sigma^*\ \forall \alpha \in out_T(w) : \alpha \models c \leq \boldsymbol{max}_A$.

The last condition in the definition of CAs is semantic and can be achieved in different ways in practice. For instance, regular expressions can be compiled to CAs where assignment terms are of the form $c + 1$, 0, or c only, and every appearance of $c + 1$

[1] To handle large or infinite sets of symbols symbolically, the predicates $\mathtt{1} = a$ may be generalised to predicates from an arbitrary effective Boolean algebra, as in [23].

is paired with a guard containing a constraint $c \leq k$ for some $k \in \mathbb{N}$. In this case, $\boldsymbol{max}_A = K + 1$ where K is the maximum constant used in the guards of the form $c \leq k$.

We will often consider the initial and final formulae of CAs given as a disjunction $\bigvee_{q \in Q}(\mathtt{s} = q \wedge \varphi_q)$ where φ_q is a formula over counter guards, in which case we write $I(q)$ or $F(q)$ to denote the disjunct φ_q of the initial or final formula, respectively. An example of a CA is given in Figure 3.1.

Deterministic counting automata. A *deterministic counting automaton* (DCA) is a CA A where I has at most one model and, for every symbol $a \in \Sigma$, every reachable configuration α has at most one a-successor (equivalently, the outcome of every word in A is either a singleton or the empty set). Finally, in the special case when $C = \emptyset$, the CA is a (classical) nondeterministic *finite automaton* (NFA), or a deterministic finite automaton (DFA) if it is deterministic.

3.3 Determinisation of Counting Automata

In this section, we discuss an algorithm for determinising CAs. A naive determinisation converts a given CA A into an NFA by hard-wiring counter configurations as a part of control states, followed by the classical subset construction to determinise the obtained NFA (the NFA is finite due to the bounds on the maximum values of counters). The state space of the obtained DFA then consists of all reachable outcomes of A. By determinising A in this way, the succinctness of using counters is lost, and the size of the DFA can explode exponentially not only in the number of control states of A but also in the number of reachable counter valuations, which makes the construction impractical. Instead, our construction will retain counters (though their number may grow) and represent possible word outcomes as configurations of the resulting DCA.

Spheres. In particular, the outcome of a word $w \in \Sigma^*$ in a CA $A = (Q, C, I, F, \Delta)$ can be represented as a formula φ over equalities of the form $c = k$ and $\mathtt{s} = q$ where $q \in Q$, $c \in C$, $k \in \mathbb{N}$. Intuitively, disjunctions can be used to obtain a single formula for the possibly many configurations reachable in A over w. For example, the outcome of the word '$\mathtt{aab}$' in Figure 3.1 is $\varphi : \mathtt{s} = q \vee (\mathtt{s} = r \wedge (c = 1 \vee c = 2))$. Generally, the outcome of '$\mathtt{aab}^i$', for $0 \leq i < k$, assuming $k > 2$, is $\varphi_i : \mathtt{s} = q \vee (\mathtt{s} = r \wedge (c = i \vee c = i + 1))$.

A crucial notion for our construction is then the notion of *sphere*. A sphere ψ arises from an outcome φ by replacing the constants from $\mathbb{N}$ by parameters drawn from a countable set $\mathcal{P}$ (disjoint from $\mathbb{N}$, V, Q, and $\{1, \mathtt{s}\}$). In the example above, the sphere obtained from the φ is $\psi : \mathtt{s} = q \vee (\mathtt{s} = r \wedge (c = p_0 \vee c = p_1))$, and the same sphere arises from all outcomes φ_i with $0 \leq i < k$.

Spheres will play the role of the control states of the resulting DCA. The idea of the construction is that the outcome of every word w in a DCA A^d will contain a single configuration (A^d is deterministic) consisting of a sphere ψ as the control state and a valuation of its parameters $\eta : \mathcal{P} \to \mathbb{N}$. The construction will ensure that $\psi[\eta]$ models the outcome $out_A(w)$ of w in A. In our example, the outcome of '$\mathtt{aab}$' in A^d would contain the single configuration $\{\mathtt{s} \mapsto \psi, p_0 \mapsto 1, p_1 \mapsto 2\}$, and the outcome of each φ_i, for $0 \leq i < k$, would contain the single configuration $\{\mathtt{s} \mapsto \psi, p_0 \mapsto i, p_1 \mapsto i + 1\}$.

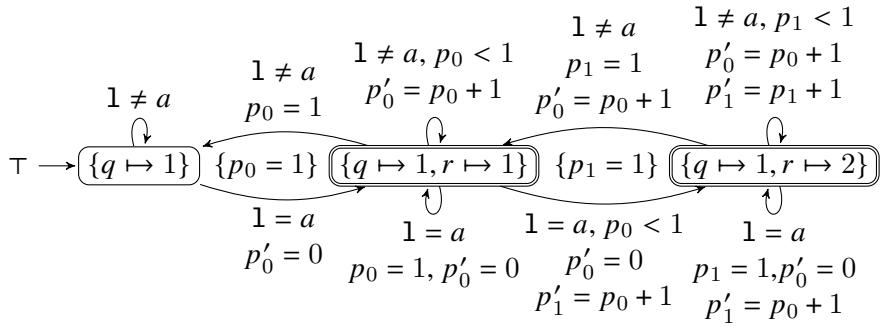


Figure 3.3: The DCA generated from the CA of Figure 3.1 for $k = 1$ by our algorithm for determinisation of monadic CA (Section 3.4).

The example shows the advantage of our construction. Every outcome φ_i would be a control state of the naively determinised automaton, with a b-transition from each φ_j to φ_{j+1}, for $0 \leq j < k - 1$. In contrast to that, all these states and transitions will be in A^d replaced by a single control state ψ with a single b-labelled self-loop that increments both p_0 and p_1. This structure can be seen in Figure 3.3 (states are spheres, labelled by their multiset representation introduced in Section 3.4).

Determinisation by sphere construction. We now provide a basic version of our sphere-based determinisation, which can also be viewed as an algorithm that constructs parametric versions of the subsets used in subset-based determinisation. For this basic algorithm, termination is not guaranteed, but it serves as a basis on which we will subsequently build a terminating algorithm. Let us first introduce some needed additional notation.

Given a formula φ, we denote by $at(\varphi)$ and by $num(\varphi)$ the sets of assignment terms and numerical constants, respectively, appearing in φ. We will use the set $\mathcal{P}' = \{p' \mid p \in \mathcal{P}\}$ and the substitution $\theta_{unprime} = \{p' \mapsto p \mid p \in \mathcal{P}\}$. We say that a formula over variables $V \cup V' \cup \{\mathtt{l}\} \cup \mathcal{P}$ is *factorised wrt guards* if it is a disjunction $\bigvee_{i=1}^{n} (g_i) \wedge (u_i)$ of factors, each consisting of a guard g_i over $V \cup \{\mathtt{l}\} \cup \mathcal{P}$ and an update formula u_i over atomic assignments such that the guards of any two different factors are mutually exclusive, i.e., $g_i \wedge g_j$ is unsatisfiable for any $1 \leq i \neq j \leq n$.[2] For a set of variables U, we denote by $\exists\!\!\!| U : \varphi$ a formula obtained by eliminating all variables in U from φ (i.e., a quantifier-free formula equivalent to $\exists U : \varphi$).[3]

[2] A Boolean combination of atomic guards and updates can be factorised through (1) a transformation to DNF, yielding a set of clauses X; (2) writing each clause $\varphi \in X$ as a conjunction of a guard formula g_φ and an assignment formula f_φ; (3) computing minterms of the set $\{g_\varphi \mid \varphi \in X\}$; (4) creating one factor $(g) \wedge (f)$ from every minterm g where f is the disjunction of all the assignment formulae f_φ with $\varphi \in X$ compatible with g (i.e., such that $g \wedge f_\varphi$ is satisfiable).

[3] We note that we only need to use a specialised, simple, and cheap quantifier elimination. In particular, we only need to eliminate counter variables c from formulae such that, in clauses of their DNF, c always appears together with a predicate $c = p$ where p is a parameter. Eliminating c from such a DNF clause is then done by simply substituting occurrences of c by p. We do not need complex algorithms such as the general quantifier elimination for Presburger arithmetic.

Algorithm 1: Sphere-based CA determinisation (non-terminating)

Input: A CA $A = (Q, C, I, F, \Delta)$.
Output: A DCA $A^d = (Q^d, P, I^d, F^d, \Delta^d)$ s.t. $L(A) = L(A^d)$.

1 $Q^d \leftarrow Worklist \leftarrow \emptyset$; $\Delta^d \leftarrow \bot$;
2 $\psi_I \leftarrow I[\theta_{const}]$ for some total injection $\theta_{const} : num(I) \to \mathcal{P}$;
3 $I^d \leftarrow \mathsf{s} = \psi_I \wedge \varphi_{\theta_{const}}$;
4 add ψ_I to Q^d and to $Worklist$;
5 **while** $Worklist \neq \emptyset$ **do**
6 $\quad \psi \leftarrow pop(Worklist)$;
7 $\quad$ Let $\bigvee_{i=1}^{n} (g_i) \wedge (u_i)$ be the formula $\exists C, \mathsf{s} : \psi \wedge \Delta$ factorised wrt guards;
8 $\quad$ **foreach** $1 \leq i \leq n$ **do**
9 $\quad\quad \psi_i \leftarrow u_i[\theta_{at}][\theta_{unprime}]$ for a total injection $\theta_{at} : at(u_i) \to \mathcal{P}'$;
10 $\quad\quad$ add $\psi \text{-}\{g_i, \varphi_{\theta_{at}}\}\!\!\to \psi_i$ to Δ^d;
11 $\quad\quad$ **if** $\psi_i \notin Q^d$ **then** add ψ_i to Q^d and to $Worklist$;
12 $P \leftarrow$ all parameters found in Q^d;
13 $F^d \leftarrow \bigvee_{\psi \in Q^d} \mathsf{s} = \psi \wedge \exists C, \mathsf{s} : \psi \wedge F$;
14 $I^d \leftarrow ground(I^d)$; $\Delta^d \leftarrow ground(\Delta^d)$;
15 **return** $A^d = (Q^d, P, I^d, F^d, \Delta^d)$;

The algorithm. The core of our determinisation algorithm is the sphere construction described in Algorithm 1. It builds a DCA $A^d = (Q^d, P, I^d, F^d, \Delta^d)$ whose control states Q^d are spheres. Its counters are parameters from the set P that is built during the run of the algorithm. The initial formula I^d defined on Line 3 assigns to s the initial control state ψ_I (obtained on Line 2), which is a parametric version of I with integer constants replaced by parameters according to the renaming θ_{const}. Moreover, I^d also equates the parameters in ψ_I with the constants they are replacing in I. Hence, the formula $\psi_I[\theta_{const}^{-1}]$ models exactly the initial configurations of A.

Example 3.3.1. In the running example (Figure 3.1), whenever referring to some variable that is assigned multiple times during the run of the algorithm, we use superscripts to distinguish the different assignments during the run. On Lines 1–4, the initial sphere ψ_I is assigned the formula $\mathsf{s} = q$, and the initial formula I^d is set to $\mathsf{s} = \psi_I$, which specifies that ψ_I is indeed the initial control state only (I does not constrain counters, hence I^d does not talk about parameters). □

The remaining states of Q^d and transitions of Δ^d are computed by a worklist algorithm on Line 5 with the worklist initialised with ψ_I. Every iteration computes the outgoing transitions of a control state $\psi \in Worklist$ as follows: On Line 7, after eliminating $C \cup \{\mathsf{s}\}$ from the formula $\psi \wedge \Delta$, which describes how the next state and counter values depend on the input symbol and the current values of parameters, it is transformed into a guard-factorised form.

Example 3.3.2. When ψ_I is taken from $Worklist$ as ψ^1 on Line 6, its processing starts by factorising $\exists \{c, \mathsf{s}\} : \psi^1 \wedge \Delta$ on Line 7. Here, $\psi^1 \wedge \Delta$ is the formula $\mathsf{s} = q \wedge (q \text{-}\{\top,\top\}\!\!\to q \vee q \text{-}\{\mathsf{l}=\mathsf{a}, c'=0\}\!\!\to r \vee r \text{-}\{c<k, c'=c+1\}\!\!\to r)$, which can be also written as:

$$\mathsf{s} = q \wedge (\mathsf{s}' = q \vee (\mathsf{l} = \mathsf{a} \wedge c' = 0 \wedge \mathsf{s}' = r)) .$$

The elimination of $\{c, \mathtt{s}\}$ gives the formula $\mathtt{s}' = q \vee (\mathtt{l} = \mathtt{a} \wedge c' = 0 \wedge \mathtt{s}' = r)$. This formula is factorised into the following two factors:

(F_1) $(\mathtt{l} = \mathtt{a}) \wedge (\mathtt{s}' = q \vee (c' = 0 \wedge \mathtt{s}' = r))$,
(F_2) $(\mathtt{l} \neq \mathtt{a}) \wedge (\mathtt{s}' = q)$. □

In the for-loop on Line 8, every factor $(g_i) \wedge (u_i)$ is turned into a transition with the guard g_i; the mutual incompatibility of the guards guarantees determinism. The formula u_i describes the target sphere in terms of the parameters of the source sphere ψ, updated according to the transition relation. That is, it is a Boolean combination of assignments of the form $c' = p + k$ or $c' = k$ for $c \in C, p \in \mathcal{P}$, and $k \in \mathbb{N}$. Line 9 creates a sphere by substituting each of the assignment terms (of the form $p + k$ or k) with a parameter and replacing primed variables by their unprimed versions.[4] The corresponding assignment term substitution θ_{at} records how the values of the new parameters are obtained from the original values of the parameters occurring in ψ. It is used to define the assignment formula of the new transition that is added to Δ^d on Line 10. The argument justifying that the construction preserves the language is the following: if reading $w \in \Sigma^*$ takes A^d to ψ with a parameter valuation η such that $\psi[\eta]$ is equivalent to $out_A(w)$, then reading a next symbol a using a transition newly created on Line 10 takes A^d to ψ' with the parameter valuation η' such that $\psi'[\eta']$ models $out_A(wa)$.

Example 3.3.3. Factor F_1 of Example 3.3.2 above is processed as follows. A possible choice for θ^1_{at} on Line 9 is the assignment $\{0 \mapsto p_0\}$. Its application followed by $\theta_{unprime}$ creates:

$$\psi^1_1 : \mathtt{s} = q \vee (c = p_0 \wedge \mathtt{s} = r).$$

From θ^1_{at}, we get the substitution formula $\varphi_{\theta^1_{at}} : (p'_0 = 0)$ on Line 10, and so the transition added to Δ^d is $(\mathtt{s} = q) \text{-}\{\mathtt{l}=\mathtt{a}, p'_0=0\}\mapsto (\mathtt{s} = q \vee (c = p_0 \wedge \mathtt{s} = r))$. The target ψ^1_1 of the transition is added to Q^d and to $Worklist$ on Line 11. Next, Factor F_2 generates the self-loop $(\mathtt{s} = q) \text{-}\{\mathtt{l}\neq\mathtt{a}, \top\}\mapsto (\mathtt{s} = q)$, which ends the first iteration of the while-loop.

Let us also walk through a part of the second iteration of the while-loop, in which ψ^1_1 is taken from $Worklist$ as ψ^2 on Line 6. The formula $\psi^2 \wedge \Delta$ from Line 7 is:

$$((\mathtt{s} = r \wedge c = p_0) \vee \mathtt{s} = q) \ \wedge \ (q \text{-}\{\top,\top\}\mapsto q \ \vee \ q \text{-}\{\mathtt{l}=\mathtt{a}, c'=0\}\mapsto r \ \vee \ r \text{-}\{c<k, c'=c+1\}\mapsto r), \tag{3.1}$$

which is equivalent to:

$$(\mathtt{s} = q \wedge (\mathtt{s}' = q \vee (\mathtt{l} = \mathtt{a} \wedge c' = 0 \wedge \mathtt{s}' = r))) \vee \big(\mathtt{s} = r \wedge c = p_0 \wedge c < k \wedge c' = c + 1 \wedge \mathtt{s}' = r\big). \tag{3.2}$$

The elimination of $\{c, \mathtt{s}\}$ on Line 7 then gives the formula:

$$\big(\mathtt{s}' = q \vee (\mathtt{l} = \mathtt{a} \wedge c' = 0 \wedge \mathtt{s}' = r)\big) \vee (p_0 < k \wedge c' = p_0 + 1 \wedge \mathtt{s}' = r), \tag{3.3}$$

which is factorised into the following four factors:

[4] The choice of the parameters in the image of $\theta_{at} : at(u_i) \to \mathcal{P}'$ on Line 9 is arbitrary, although, in practice, it would be sensible to define some systematic parameter naming policy and reuse existing parameters whenever possible.

(F_3) $(\mathtt{l} = \mathtt{a} \wedge p_0 < k) \wedge (\mathtt{s}' = q \vee (c' = 0 \wedge \mathtt{s}' = r) \vee (c' = p_0 + 1 \wedge \mathtt{s}' = r))$,
(F_4) $(\mathtt{l} \neq \mathtt{a} \wedge p_0 < k) \wedge (\mathtt{s}' = q \vee (c' = p_0 + 1 \wedge \mathtt{s}' = r))$,
(F_5) $(\mathtt{l} = \mathtt{a} \wedge p_0 \geq k) \wedge (\mathtt{s}' = q \vee (c' = 0 \wedge \mathtt{s}' = r))$, and
(F_6) $(\mathtt{l} \neq \mathtt{a} \wedge p_0 \geq k) \wedge (\mathtt{s}' = q)$.

In the for-loop on Line 8, Factor F_3 is processed as follows. Let the chosen substitution θ^2_{at} on Line 9 be $\{p_0{+}1 \mapsto p_1, 0 \mapsto p_0\}$. Its application followed by $\theta_{unprime}$ generates:

$$\psi^2_1 : \mathtt{s} = q \vee (c = p_0 \wedge \mathtt{s} = r) \vee (c = p_1 \wedge \mathtt{s} = r).$$

The substitution formula $\varphi_{\theta^2_{at}}$ on Line 10 is $p'_1 = p_0 + 1 \wedge p'_0 = 0$, and so Δ^d gets the new transition $\psi^1_1 \text{-}\{\mathtt{l}=\mathtt{a} \wedge p_0 < k, p'_1 = p_0+1 \wedge p'_0 = 0\}\text{-}\!\!\rightarrow \psi^2_1$. The evaluation of the while-loop would continue analogously. □

In the final stage of the algorithm, when (and if) the while-loop terminates, Line 12 collects the set P of all parameters used in the constructed parametric spheres of Q^d as new counters of A^d. Further, Line 13 derives the new final formula by considering all computed spheres, restricting them to valuations where the original final formula is satisfied, and quantifying out the original counters. This way, final constraints on the original counters get translated to constraints over parameters in P.

Example 3.3.4. In our running example, for the spheres discussed above, we would have $F(\psi^1) : \bot$, $F(\psi^1_1) : p_0 = 1$, and $F(\psi^2_1) : p_0 = 1 \vee p_1 = 1$. □

Finally, Line 14 applies the function $ground$ on the initial formula and the transition formula of the constructed automaton before returning it. This step is needed in order to avoid nondeterminism on unused and unconstrained counters. The function $ground$ conjuncts constraints of the form $p = 0$ with the initial formula and with the guard of every transition for every parameter $p \in P$ that is so far unconstrained in the concerned formula. Moreover, it will introduce a reset $p' = 0$ to the assignment formula of every transition for every counter $p \in P$ that is so far not assigned on the concerned transition. The while-loop of Algorithm 1 needs, however, not terminate, as witnessed also by our example.[5]

Example 3.3.5. Continuing in Example 3.3.4, the DCA in Figure 3.3 would be a part of the DCA constructed by Algorithm 1, its states being the spheres ψ^1, ψ^1_1, ψ^2_1 from the left, but the while-loop would not terminate, with ψ^2_1. Instead, it would eventually generate a successor of ψ^2_1, the sphere:

$$\psi^3_1 : \mathtt{s} = q \vee (c = p_0 \wedge \mathtt{s} = r) \vee (c = p_1 \wedge \mathtt{s} = r) \vee (c = p_2 \wedge \mathtt{s} = r),$$

i.e., a sphere similar to ψ^2_1 but extended by a new disjunct with a new parameter p_2. Repeating this, the algorithm would keep generating larger and larger spheres with more and more parameters. □

[5]For this step to preserve the language of the automaton, we need to assume that the input CA does not assign nondeterministic values to live counters. We are refering to the standard notion: a counter is live at a state if the value it holds at that state may influence satisfaction of some guard in the future. Any CA can be transformed into this form, and CAs we compile from regular expressions satisfy this condition by construction.

Example 3.3.6. We illustrate the way Algorithm 1 works on the CA presented in Figure 3.1. Whenever referring to some variable that is assigned multiple times during the run of the algorithm, we use superscripts to distinguish the different values of the variable. The algorithm first gives us $\psi_I \stackrel{\text{def}}{=} (\mathtt{s} = q)$ and $I^d \stackrel{\text{def}}{=} (\mathtt{s} = \psi_I)$. When ψ_I is taken from *Worklist* as ψ^1, its processing starts by factorising $\exists\!\!\exists\{c, \mathtt{s}\} : \psi^1 \wedge \Delta$. Here, $\psi^1 \wedge \Delta$ is the formula $\mathtt{s} = q \wedge [q \text{-}\{\top,\top\}\!\!\rightarrow q \vee q \text{-}\{\mathtt{l}=a, c'=0\}\!\!\rightarrow r \vee r \text{-}\{c<k, c'=c+1\}\!\!\rightarrow r]$, which can be rewritten to the formula $\mathtt{s} = q \wedge (\mathtt{s}' = q \vee (\mathtt{l} = a \wedge c' = 0 \wedge \mathtt{s}' = r))$. The elimination of $\{c, \mathtt{s}\}$ gives the formula $\mathtt{s}' = q \vee (\mathtt{l} = a \wedge c' = 0 \wedge \mathtt{s}' = r)$. This formula can be factorised to the following two factors: $(\mathtt{l} = a) \wedge (\mathtt{s}' = q \vee (c' = 0 \wedge \mathtt{s}' = r))$ and $(\mathtt{l} \neq a) \wedge (\mathtt{s}' = q)$.

We start by processing the first factor. A possible choice for θ^1_{at} is the assignment $\{0 \mapsto p_1\}$. Its application followed by $\theta_{unprime}$ creates $\psi^1_1 \stackrel{\text{def}}{=} (\mathtt{s} = q \vee (c = p_1 \wedge \mathtt{s} = r))$. From θ^1_{at}, we get the substitution formula $\varphi_{\theta^1_{at}} \stackrel{\text{def}}{=} (p'_1 = 0)$, and so the transition added to Δ^d is $(\mathtt{s} = q) \text{-}\{\mathtt{l}=a\ , p'_1=0\}\!\!\rightarrow (\mathtt{s} = q \vee (c = p_1 \wedge \mathtt{s} = r))$. The target of the transition is added to Q^d and to *Worklist*. The other factor generates the self-loop $(\mathtt{s} = q) \text{-}\{\mathtt{l}\neq a, \top\}\!\!\rightarrow (\mathtt{s} = q)$.

The algorithm goes on by processing ψ^1_1 taken from *Worklist* as ψ^2. In this case, the formula $\psi^2 \wedge \Delta$ is $((\mathtt{s} = r \wedge c = p_1) \vee \mathtt{s} = q) \wedge [q \text{-}\{\top,\top\}\!\!\rightarrow q \vee q \text{-}\{\mathtt{l}=a, c'=0\}\!\!\rightarrow r \vee r \text{-}\{c<k, c'=c+1\}\!\!\rightarrow r]$, which is equivalent to the formula $(\mathtt{s} = q \wedge (\mathtt{s}' = q \vee (\mathtt{l} = a \wedge c' = 0 \wedge \mathtt{s}' = r))) \vee (\mathtt{s} = r \wedge c = p_1 \wedge c < k \wedge c' = c+1 \wedge \mathtt{s}' = r)$. The elimination of $\{c, \mathtt{s}\}$ then gives the formula $(\mathtt{s}' = q \vee (\mathtt{l} = a \wedge c' = 0 \wedge \mathtt{s}' = r)) \vee (p_1 < k \wedge c' = p_1 + 1 \wedge \mathtt{s}' = r)$. This formula can be factorised to the following four factors:

- $(\mathtt{l} = a \ \wedge p_1 < k) \wedge (\mathtt{s}' = q \vee (c' = 0 \wedge \mathtt{s}' = r) \vee (c' = p_1 + 1 \wedge \mathtt{s}' = r))$,
- $(\mathtt{l} \neq a \ \wedge p_1 < k) \wedge (\mathtt{s}' = q \vee (c' = p_1 + 1 \wedge \mathtt{s}' = r))$,
- $(\mathtt{l} = a \ \wedge p_1 \geq k) \wedge (\mathtt{s}' = q \vee (c' = 0 \wedge \mathtt{s}' = r))$, and
- $(\mathtt{l} \neq a \ \wedge p_1 \geq k) \wedge (\mathtt{s}' = q)$.

We proceed by processing the first factor. A possible choice for the substitution θ^2_{at} is $\{p_1 + 1 \mapsto p_1, 0 \mapsto p_2\}$. Its application followed by $\theta_{unprime}$ generates:

$$\psi^2_1 : (\mathtt{s} = q \vee (c = p_1 \wedge \mathtt{s} = r) \vee (c = p_2 \wedge \mathtt{s} = r)) .$$

The substitution formula $\varphi_{\theta^2_{at}}$ is: $(p'_1 = p_1 + 1 \wedge p'_2 = 0)$, so the transition added to Δ^d is

$$\psi^1_1 \text{-}\{\mathtt{l}=a \wedge p_1<k, p'_1=p_1+1 \wedge p'_2=0\}\!\!\rightarrow \psi^2_1.$$

Later on, Algorithm 1 would in a similar manner from ψ^2_1 generate the sphere:

$$\psi^3_1 : (\mathtt{s} = q \vee (c = p_1 \wedge \mathtt{s} = r) \vee (c = p_2 \wedge \mathtt{s} = r) \vee (c = p_3 \wedge \mathtt{s} = r)) ,$$

i.e., a sphere similar to ψ^2_1 but extended by a new disjunct with a new parameter p_3. When continued, more and more parameters would be introduced, obtaining bigger and bigger spheres, and the algorithm would never terminate. The reasons why this nontermination happens and a way how to tackle it are discussed in the next section. □

Ensuring termination of the sphere construction. In this section, we will discus reasons for possible non-termination of Algorithm 1 and a way to tackle them. The main reason is that the algorithm may generate unboundedly many parameters that correspond to different histories of a counter c when processing the input word (including also impossible ones in which the counter exceeds the maximum value). The algorithm indeed "splits" a parameter appearing in a sphere into two parameters in the successor sphere when the transitions of A update the counter in two different ways.

In our terminating version of Algorithm 1, we build on the following: (1) distinguishing between histories that converge in the same counter value is not necessary, they can be "merged", and (2) the number of different reachable counter values is bounded (by the definition of CAs). We thus enforce the invariant of every reachable configuration of A^d that all parameters in the configuration have distinct values. The invariant is enforced by testing equalities of parameters and merging parameters with equal values on transitions of A^d. All transitions of A^d entering spheres with more than $\boldsymbol{max}_A + 1$ parameters can then be discarded because the invariant implies that they cannot be taken at any configuration of A^d. Furthermore, we will also ensure that the algorithm does not diverge because of generating semantically equivalent but syntactically different spheres (because of different names of parameters or different formulae structure).

A terminating determinisation of CAs is obtained from Algorithm 1 by replacing Lines 9–11 by the code in Algorithm 2. In order to ensure that parameters have pairwise distinct values, the transitions of A^d test equalities of the values assigned to parameters and ensure that two parameters are never used to represent the same value. Different histories of counters are thus merged if they converge into the same value. To achieve this, Algorithm 2 enumerates all feasible equivalences of the assignment terms of u_i on Line 16 and generates successor transitions for each of them separately. When deciding whether an equivalence $\sim$ on the assignment terms is feasible, the algorithm performs two tests:

1. The formula:

$$\varphi_{\sim} \stackrel{\text{def}}{=} \bigwedge_{t_1 \sim t_2, t_1, t_2 \in at(u_i)} (t_1 = t_2) \wedge \bigwedge_{t_1 \not\sim t_2, t_1, t_2 \in at(u_i)} (t_1 \neq t_2)$$

   is tested for satisfiability, meaning that the equivalence is not trying to merge terms that can never be equal (such as, e.g., p and $p + 1$).

2. The number of equivalence classes should be at most $\boldsymbol{max}_A + 1$ since this is the maximum number of different values that the counters can reach due to the requirement that the values must be between 0 and $\boldsymbol{max}_A$.

Line 17 builds a term assignment replacement θ_{at} that maps all $\sim$-equivalent terms to the same (future) parameter, and Line 18 computes the target sphere, reflecting the given merge. The test on Line 19 checks whether the target sphere is equal to some already generated sphere up to a parameter renaming (represented by a bijection $\theta_{rename} : \mathcal{P} \leftrightarrow \mathcal{P}$). If so, the created sphere is discarded, and a new transition going to the old sphere is generated on Line 20; we need to rename the primed parameters used in the transition's assignment appropriately according to

$$\theta'_{rename} = \{p'_0 \mapsto p'_1 \mid p_0 \mapsto p_1 \in \theta_{rename}\}.$$

Algorithm 2: Ensuring termination of sphere-based CA determinisation

16 **foreach** *equivalence* $\sim$ *on* $at(u_i)$ *s.t.* $\texttt{sat}(\varphi_\sim)$ *and* $|at(u_i)/_\sim| \leq \boldsymbol{max}_A + 1$ **do**
17 **let** $\theta_{at} : at(u_i) \to \mathcal{P}'$ be an injection;
18 $\psi_i \leftarrow u_i[\theta_{at}][\theta_{unprime}]$;
19 **if** $\exists \theta_{rename} : \mathcal{P} \leftrightarrow \mathcal{P}\ \exists \sigma \in Q^d : \psi_i[\theta_{rename}] \Leftrightarrow \sigma$ **then**
20 add $\psi \text{-}\{g_i \wedge \varphi_\sim[\theta_{at}]\varphi_{\theta_{at}}[\theta'_{rename}]\}\!\!\to, \sigma$ to Δ^d;
21 **else**
22 add $\psi \text{-}\{g_i \wedge \varphi_\sim[\theta_{at}], \varphi_{\theta_{at}}\}\!\!\to \psi_i$ to Δ^d;
23 add ψ_i to Q^d and to $Worklist$;

Otherwise, a transition into the new sphere is added on Line 22, and the new sphere is added to Q^d and $Worklist$. In both cases, the guard of the generated transition is extended by the formula $\varphi_\sim[\theta_{at}]$, which encodes the equivalence $\sim$, and hence explicitly enforces that $\sim$ holds when the transition is taken.

Note that the test on the maximum number of equivalence classes can be optimised if finer information about the maximum reachable values of the individual counters is available. Such information can be obtained, e.g., by looking at the constants used in the guards of the transitions where the different counters are increased. For any counter, one should then not generate more parameters representing its possible values than what the upper bound on that counter is (plus one).

Theorem 1. *Algorithm 1 with the modification presented in Algorithm 2 terminates and produces a DCA with* $L(A) = L(A^d)$ *and* $|Q^d| \leq 2^{|Q| \cdot (\boldsymbol{max}_A + 1)^{|C|}}$.

Proof (sketch). The fact that the algorithm indeed constructs a DCA is because Line 7 of Algorithm 1 generates pairwise incompatible guards on transitions only. It is also easy to show by induction on the length of the words that the language is preserved. The termination then follows from the facts that (1) the algorithm has a bound on the maximum number of parameters in spheres (ensured by the condition over $\sim$ on Line 16 of Algorithm 2) and (2) no spheres equal up to renaming are generated (ensured by the check on Line 19). The bound on the size follows from the structure of spheres. □

The number of equivalences generated on Line 16 of Algorithm 2 (and therefore also the number of transitions leaving ψ) may be large. Many of them are, however, infeasible (cannot be taken in any reachable configuration of A^d), and could be removed. In most cases, the majority of such infeasible transitions may be identified locally, taking advantage of the invariant of all reachable configurations of A^d enforced by Algorithm 2: namely, values of distinct parameters are always pairwise distinct. Therefore, before building a transition for an equivalence $\sim$, we ask whether the $\sim$-equivalent assignment terms may indeed be made equivalent assuming that the constructed transition guard g_i and—importantly—also the distinctness invariant hold right before the transition is taken. Technically, we create new transitions only from those equivalences $\sim$ such that:

$$\texttt{sat}\Big(\bigwedge_{p_1, p_2 \in P_\psi, dist(p_1, p_2)} (p_1 \neq p_2) \wedge g_i \wedge \varphi_\sim\Big)$$

where P_ψ is the set of parameters of ψ and $dist(p_1, p_2)$ holds iff p_1 and p_2 are distinct parameters.

Example 3.3.7. From the CA in Figure 3.1, Algorithm 2 would generate the DCA shown in Figure 3.3, with the addition of the parameter equivalence tests on the self-loops on the right-most state. Each of the self-loops would exist in two variants, with $p'_0 = p'_1$ and $p'_0 \neq p'_1$. The optimisation that takes advantage of the distinctness invariant discussed above would prune away the possibilities with $p'_0 = p'_1$, hence the generated DCA would be identical to the one in Figure 3.3 up to the additional guard $p'_0 \neq p'_1$ on the self-loops of the right-most state. □

Reachability-restricted CA determinisation. Above, we have described a terminating algorithm for CA determinisation. While it is witnessed by our experiments that the algorithm often generates much smaller automata than what could be obtained by transforming the automata into NFAs and determinising them, a natural question is whether the generated DCA is *always* smaller or equal in size to the DFA built by getting rid of the counters and using classical determinisation. Unfortunately, the answer to this question is no. The reason is that the transformation to a DCA needs not recognise that some generated transitions can never be executed and that some spheres are not reachable. To see this, it is enough to imagine a transition setting some counter c to zero and the only successor transition testing whether c is positive. The latter transition would not be executed when generating the DFA due to working with concrete values of counters, but it would be considered when constructing the DCA (since the construction does not know the values of the counters).

In our experiments with CAs obtained from real-life regexes, the above was not a problem, but we note that, for the price of an increased cost of the construction, one could further improve the algorithm by taking into account some reachability information. In an extreme case, one could first generate the DFA corresponding to the given CA and then use it when generating the DCA (as a hopefully more compact representation of the DFA). In particular, whenever adding some new sphere into the DCA being built, the algorithm can check whether there is a subset of states in the original CA represented as a state of the DFA that is an instance of the sphere. If not, the sphere is not added. The resulting DCA can then never be bigger than the DFA since each control state of the DFA (i.e., a subset of states of the original CA) is represented by a single sphere only, likewise each transition of the DFA is represented by a single transition of the DCA, and there are not any unreachable spheres or transitions that cannot be executed.

Notice that the reachability pruning is an alternative to Algorithm 2. Algorithm 1 equipped with the reachability analysis is guaranteed to terminate. For example, when run on the CA in Figure 3.1, it would generate a DCA isomorphic to that from Figure 3.3.

3.4 Monadic Counting

We now provide a simplified and more efficient version of the determinisation algorithm. The simplified version targets CAs that naturally arise from *monadic regexes*, i.e., regular expressions extended with counting limited to *character classes*. Their abstract syntax is:

$$R ::= \emptyset \mid \varepsilon \mid \sigma \mid R_1 R_2 \mid R_1 \mid R_2 \mid R^* \mid \sigma\{n, m\}$$

where σ is a predicate denoting a set of alphabet symbols, i.e., a *character class* (σ will be used to denote character classes from now on), and $n, m \geq 0$ are integers. The semantics is defined as usual, with $\sigma\{n, m\}$ denoting a string w with $n \leq |w| \leq m$ symbols satisfying σ.

The specialised determinisation algorithm is of a high practical relevance since the monadic class is very common, as witnessed by our experiments, where it covers over 95 % of the regexes with counting that we found (cf. Section 3.5).

Monadic counting automata. Monadic regexes can be easily compiled to nondeterministic monadic CAs satisfying certain structural properties summarised below.[6] In particular, a (nondeterministic) *monadic counting automaton (MCA)* is a CA $A = (Q, C, I, F, \Delta)$ where the following holds:

1. The set Q of control states is partitioned into a set of *simple states* Q_s and a set of *counting states* Q_c, i.e., $Q = Q_s \uplus Q_c$.

2. The set of counters $C = \{c_q \mid q \in Q_c\}$ consists of a unique counter c_q for every counting state $q \in Q_c$.

3. All transitions containing counter guards or updates must be incident with a counting state in the following manner. Every counting state $q \in Q_c$ has a single *increment transition*, a self-loop $q \text{-}\{\sigma \wedge c_q < \boldsymbol{max}_q, c'_q = c_q + 1\}\!\!\rightarrow q$ with the value of c_q limited by the *bound* $\boldsymbol{max}_q$ of q, and possibly several *entry transitions* of the form $r \text{-}\{\bar{\sigma} \wedge c'_q = 0\}\!\!\rightarrow q$, which set c_q to 0. As for *exit transitions*, every counting state is either *exact* or *range*, where exact counting states have exit transitions of the form $q \text{-}\{\sigma \wedge c_q = \boldsymbol{max}_q\}\!\!\rightarrow s$, and *range* counting states have exit transitions of the form $q \text{-}\{\sigma, \top\}\!\!\rightarrow s$ with $s \in Q$ s.t. $s \neq q$. That is, an exact counting state may be left only after exactly $\boldsymbol{max}_q$ repetitions of the incrementing transition (it corresponds to a regular expression 'σ{k}'), while a range counting state may be left sooner (it corresponds to a regular expression 'σ{0,k}'). We denote the set of range counting states Q_r and the set of exact counting states Q_e, with $Q_c = Q_r \uplus Q_e$.

4. The initial condition I is of the form

$$I : \bigvee_{q \in Q_s^I} \mathsf{s} = q \vee \bigvee_{q \in Q_c^I} (\mathsf{s} = q \wedge c_q = 0)$$

   for some sets of initial simple and counting states $Q_s^I \subseteq Q_s$ and $Q_c^I \subseteq Q_c$, respectively, with the counters of initial counting states initialised to 0.

5. The final condition F is of the form

$$F : \bigvee_{q \in Q_s^F \cup Q_r^F} \mathsf{s} = q \vee \bigvee_{q \in Q_e^F} (\mathsf{s} = q \wedge c_q = \boldsymbol{max}_q)$$

[6] We note that we restrict ourselves to range sub-expressions of the form '$\sigma\{n, n\}$' or '$\sigma\{0, n\}$' only. This is without loss of generality since a general range expression '$\sigma\{m, n\}$' can be rewritten as '$\sigma\{m, m\}.\sigma\{0, n - m\}$'.

where $Q_s^F \subseteq Q_s$ is a set of simple final states, $Q_r^F \subseteq Q_r$ is a set of final range counting states, and $Q_e^F \subseteq Q_e$ is a set of final exact counting states. That is, final conditions on final states are the same as counter conditions on exit transitions.[7]

Determinisation of MCAs. Algorithm 2 can be simplified when specialised to monadic CAs. The simplification is based on the following observations. *Observation 1. Counters are dead outside their states.* To simplify the representation of spheres, we use the fact that every counter c_q of an MCA is "active" in the state q only, while c_q is "dead" in other states (i.e., its current value has no influence on runs of the MCA that are not in q). To represent different variants of c_q, we use parameters of the form $c_q[i]$ obtained by indexing c_q by an index i, for $0 \leq i \leq \boldsymbol{max}_q$, while enforcing the invariant that, for distinct indices i and j, $c_q[i]$ and $c_q[j]$ always have different values. Since the value of c_q ranges from 0 to $\boldsymbol{max}_q$, at most $\boldsymbol{max}_q + 1$ variants of c_q are needed.[8] Since spheres only need parameters to remember values of live counters, every sphere can be equivalently written in the *normal form*:

$$\psi \stackrel{\text{def}}{=} \bigvee_{q \in Q'_s} \mathsf{s} = q \;\vee\; \bigvee_{q \in Q'_c} \Big(\mathsf{s} = q \wedge \bigvee_{0 \leq i \leq \boldsymbol{max}'_q} c_q = c_q[i] \Big)$$

for some $Q'_s \subseteq Q_s$, $Q'_c \subseteq Q_c$, and $\boldsymbol{max}'_q \leq \boldsymbol{max}_q$. That is, a sphere ψ records which states may be reached in the original MCA when ψ is reached in the determinised MCA and also which variants of the counter c_q may record the value of c_q when q is reached.

Observation 2. Variants of exact counting states can be sorted. For dealing with any exact counting state $q \in Q_e$, we may use the following facts: (1) If executed, the increment transition of q increments all variants of c_q whose values are smaller than $\boldsymbol{max}_q$. (2) New variants of c_q are initialised to 0 by the entry transitions. (3) Variants whose value is $\boldsymbol{max}_q$ can take an exit transition, after which they become dead and their values do not need to be propagated to the next configuration. It is therefore easy to enforce that the values of the variants $c_q[i]$ stay sorted, so that $i < j$ implies $\alpha(c_q[i]) < \alpha(c_q[j])$ in every configuration α of A^d. The sortedness invariant implies that the variant of c_q with the highest index, called *highest variant*, has the highest value. This, together with the invariant of boundedness by $\boldsymbol{max}_q$ and mutual distinctness of values of variants of c_q, means that the highest variant is the only one that may satisfy the tests $c_q = \boldsymbol{max}_q$ on exit transitions or fail the test $c_q < \boldsymbol{max}_q$ on the incrementing transition. Hence, the deterministic MCA does not need to test all variants of c_q but the highest one only.

Observation 3. Only the smallest variants of range counting states are important. For range counting states, we adapt the *simulation pruning* technique from [39]. The technique optimizes the standard subset-construction-based determinisation of NFAs by exploiting a *simulation* relation [29] such that any *macrostate* (which has the form of a set of states of the original NFA) obtained during the determinisation can

[7] Notice that the guards $c_q < \boldsymbol{max}_q$ on the incrementing self-loops of exact counting states could be removed without affecting the language since when c_q exceeds $\boldsymbol{max}_q$, then the run can never leave q and has thus no chance of accepting. We include these guards only to conform to the condition on boundedness of counter values in the definition of CAs.

[8] Notice that maintaining a fixed association of a parameter to a counter is a difference from Algorithms 1 and 2, where one parameter may represent different counters.

be pruned by removing those NFA states that are simulated by other NFA states included in the same macrostate. The pruning does not change the language: the resulting DFA is bisimilar to the one constructed without pruning. For our DCA construction, we use the simulation that implicitly exists between configurations α and α' of A with the same range counting state $q = \alpha(\mathtt{s}) = \alpha'(\mathtt{s})$, where $\alpha(c_q) \geq \alpha'(c_q)$ implies that α' simulates α.[9] Hence, the spheres only need to remember the smallest possible counter value for every range counting state q, which may be always stored in $c_q[0]$, and discard all other variants.

Observations 1–3 above allow for representing spheres using a simple data structure, namely, a multiset of states. By a slight abuse of notation, we use ψ for the sphere itself as well as for its multiset representation $\psi : Q \to \mathbb{N}$. The fact that $\psi(q) > 0$ means that q is present in the sphere (i.e., $\mathtt{s} = q$ is a predicate in the normal form of ψ), and for a counting state q, the counters $c_q[0], \ldots, c_q[\psi(q) - 1]$ are the $\psi(q)$ variants of c_q tracked in the sphere (i.e., $\psi(q) - 1 = \boldsymbol{max}'_q$ in the normal form of ψ).

The MCA determinisation is then an analogy of Algorithm 1 that uses the multiset data structure and preserves the sortedness and uniqueness of variants of exact counters. The initial sphere ψ_I assigns 1 to all initial states of I, and the initial configuration I^d assigns 0 to $c_q[0]$ for each counting state q in I. Further, we modify the part of Algorithm 1 after popping a sphere ψ from $Worklist$ in the main loop (Lines 7–11).

Let Δ_ψ denote the set of transitions of A originating from states q with $\psi(q) > 0$. Processing of ψ starts by removing guard predicates of the form $c_q < \boldsymbol{max}_q$ from increment transitions of exact counting states in Δ_ψ (since they have no semantic effect as mentioned already above). Subsequently, we compute minterms of the set of guard formulae of the transitions in Δ_ψ. Each minterm μ then gives rise to a transition $\psi \text{-}\{g,f\}\!\!\mapsto \psi'$ of A^d. The guard formula g, assignment formula f, and the target sphere ψ' are constructed as follows.

First, the guard g is obtained from the minterm μ by replacing, for all $q \in Q_c$, every occurrence of c_q by $c_q[\psi(q)]$, i.e., the highest variant of c_q. Intuitively, the counter guards of transitions of Δ_ψ present in μ will on the constructed transition of A^d be testing the highest variants of the counters. This is justified since (a) only the highest variant of c_q needs to be tested for exact counting states, as concluded in Observation 2 above, and (b) we keep only a single variant of c_q for range counting states (which is also the highest one), as concluded in Observation 3.

We then initialise the target multiset ψ' as the empty multiset $\{q \mapsto 0 \mid q \in Q\}$ and collect the set Δ_μ of all transitions from Δ_ψ that are compatible with the minterm μ (recall that increment self-loops of exact states in Δ_ψ have counter guards removed, hence counter guards do not influence their inclusion in Δ_μ). The transitions of Δ_μ will be processed in the following three steps.

Step 1 (simple states). Simple states with an incoming transition in Δ_μ get $\psi'(q) = 1$.

Step 2 (increment self-loops). For exact states with the increment self-loop in Δ_μ, $\psi'(q)$ is set to $\psi(q) - 1$ if an exit transition of q is in Δ_μ, and to $\psi(q)$ otherwise. Indeed, if (and only if) an exit transition of q is included in Δ_μ, and Δ_μ is enabled in some sphere, then the highest variant of c_q has reached $\boldsymbol{max}_q$ in that sphere, and the self-loop

[9] The fact that this relation is indeed a simulation can be seen from that both the higher and lower value of c_q can use any exit transition of q at any moment regardless of the value of c_q, but the lower value of c_q can stay in the counting loop longer.

cannot be taken by the highest variant of c_q. The lower variants of c_q always have values smaller than $\boldsymbol{max}_q$, and hence can take the self-loop. The assignment f then gets the conjunct $c_q[i]' = c_q[i] + 1$ for each $0 \leq i < \psi'(q)$ since the variants that take the self-loop are incremented. For range states with the increment self-loop in Δ_μ, we set $\psi'(q)$ to 1, and $c_q[0]' = c_q[0] + 1$ is added to f (only one variant is remembered).

Step 3 (entry transitions). For each counting state q with an entry transition in Δ_μ, $\psi'(q)$ is incremented by 1 and the assignment $c_q[0]' = 0$ of the fresh variant of c_q is added to f. If the new value of $\psi'(q)$ exceeds $\boldsymbol{max}_q + 1$, then the whole transition generated from μ is discarded, since c_q cannot have more than $\boldsymbol{max}_q + 1$ distinct values. Otherwise, if q is an exact counting state, then f is updated to preserve the invariant of sorted and unique values of c_q: the increments of older variants of c_q are *right-shifted* to make space for the fresh variant, meaning that each conjunct $c_q[i]' = c_q[i] + 1$ in f is replaced by $c_q[i+1]' = c_q[i] + 1$. If $q \in Q_r$, then if the assignment $c_q[0]' = c_q[0] + 1$ is present in f, it is removed (as the fresh variant has the smallest value 0).

Example 3.4.1. Determinising the CA from Figure 3.1 using the algorithm described in this section would result in the DCA shown in Figure 3.3. □

The monadic determinisation has a much lower worst-case complexity than the general algorithm. Importantly, the number of states depends on $\boldsymbol{max}_A$ only polynomially, which is a major difference from the exponential bounds of the naive determinisation and our general construction.

Theorem 2. *The specialised monadic CA determinisation constructs a DCA with* $|Q^d| \leq (\boldsymbol{max}_A + 1)^{|Q|}$ *and* $|\Delta^d| \leq |\Sigma| \cdot (4 \cdot (\boldsymbol{max}_A + 1))^{|Q|}$.

Proof (sketch). The bound on the number of states is given by the number of functions $Q \to \{0, \ldots, \boldsymbol{max}_A\}$. The bound on the number of transitions is given by the fact, that if a sphere multiset maps a state q to n, then the successors of the sphere can map q to 0 (when q is not a successor), $n - 1$, n, or $n + 1$. Therefore, for every symbol from Σ and every macrostate from at most $(\boldsymbol{max}_A + 1)^{|Q|}$ many of them, there are at most $4^{|Q|}$ successors, and $|\Sigma| \cdot (\boldsymbol{max}_A + 1)^{|Q|} \cdot 4^{|Q|} = (4 \cdot (\boldsymbol{max}_A + 1))^{|Q|}$. □

3.5 Experimental Evaluation

The main purpose of our experimentation was to compare the proposed approach with the naive determinisation and confirm that our method produces significantly smaller automata and mitigates the risk of the state space explosion causing a complete failure of determinisation (and the implied impossibility to use the desired deterministic automaton for the intended application, such as pattern matching). To this end, we extended the Microsoft's Automata library [67] with a prototype support for CAs, implemented the algorithm from Section 3.4 (denoted **Counting** in the following), and compared it to the standard determinisation already present in the library (denoted as **DFA**). For the evaluation, we collected 2,361 regexes from a wide range of applications—namely, those used in network intrusion detection systems (Snort [63]: 741 regexes, Yang [110]: 228 regexes, Bro [80]: 417 regexes, HomeBrewed [102]: 55 regexes), the Microsoft's security leak scanning system (Industrial: 17 regexes), the Sagan log analysis engine (Sagan [95]: 14 regexes), and the pattern matching rules from

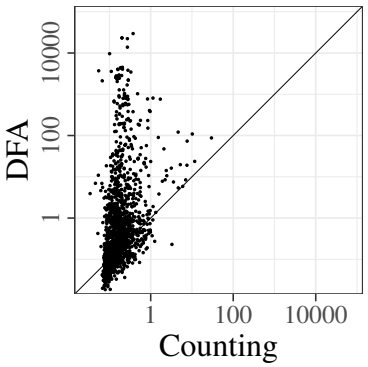


Figure 3.4: Comparison of running times given in ms (the axes are logarithmic).

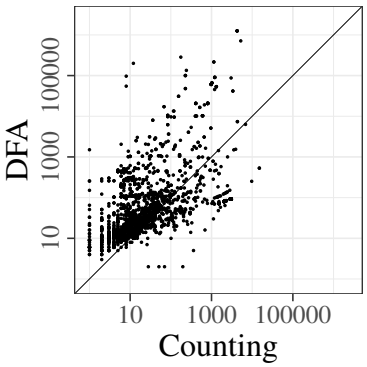


Figure 3.5: Comparison of numbers of states (the axes are logarithmic).

RegExLib (RegExLib [78]: 889 regexes). We only selected regexes that contain an occurrence of the counting operator, and from these, we selected only monadic ones (there were over 95 % of them, confirming the fragment's importance). All benchmarks were run on a Xeon E5-2620v2@2.4GHz CPU with 32 GiB RAM with a timeout of 1 min (we take the mean time of 10 runs). In the following, we use μ, m, and σ to denote the statistical indicators mean, median, and standard deviation, respectively. All times are reported in milliseconds.

The number of timeouts was 110 for **Counting**, and 238 for **DFA**. The two methods were to some degree complementary, there were only 62 cases in which both timed out. This confirms that our algorithm indeed mitigates the risk of failure due to state space explosion in determinisation. The remaining comparisons are done only with respect to benchmarks for which neither of the methods timed out.

In Figure 3.4, we compare the running times of the conversion of an NFA for a given regex to a DFA (the **DFA** axis) and the determinisation of the CA for the same regex (the **Counting** axis). If we exclude the easy cases where both approaches finished within 1 ms, we can see that **Counting** is almost always better than **DFA**. Note that the axes are logarithmic, so the advantage of **Counting** over **DFA** grows exponentially wrt the distance of the data point from the diagonal. The statistical indicators for the running times are $\mu = 110$, $m = 0.17$, $\sigma = 1,177$ for **DFA** and $\mu = 0.23$, $m = 0.13$, $\sigma = 0.09$ for **Counting**.

In Figure 3.5, we compare the number of states of the results of the determinisation algorithms (DCA for **Counting** and DFA for **DFA**). Also here, **Counting** significantly dominates **DFA**. The statistical indicators for the numbers of states are $\mu = 4,543$, $m = 41$, $\sigma = 57,543$ for **DFA** and $\mu = 241$, $m = 13$, $\sigma = 800$ for **Counting**. To better evaluate the conciseness of using DCAs, we further selected 184 benchmarks that suffered from state explosion during determinisation (our criterion for the selection was that the number of states increased at least ten-fold in **DFA**) and explored how the CA model can be used to mitigate the explosion. Figure 3.6 shows histograms of how DCAs were more compact than DFAs and also how much the number of counters rose during the determinisation. From the histograms, we can see that there are indeed many cases where the use of DCAs allows one to use a significantly more compact

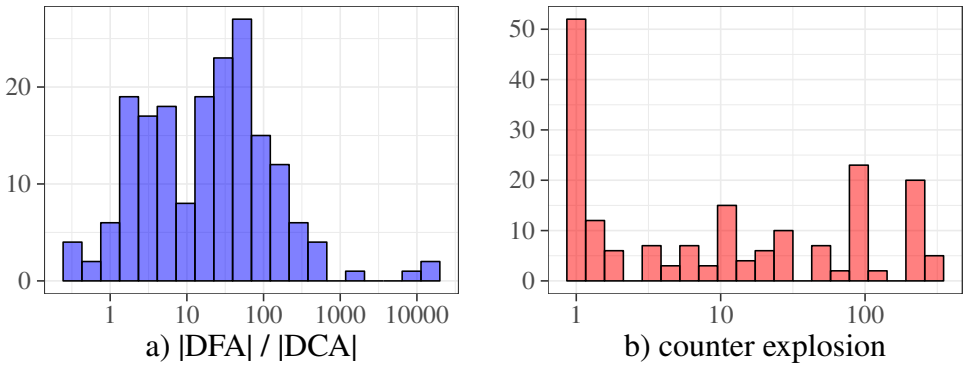


Figure 3.6: Histograms of (a) the ratio of the number of states of a DFA and of the corresponding DCA (i.e., a bar at value x of a height h denotes that the size of the DCA was h times around x times smaller than the size of the corresponding DFA) and (b) the ratio of the number of counters used by a CA after and before determinisation. Note that the x-axes are logarithmic in both cases.

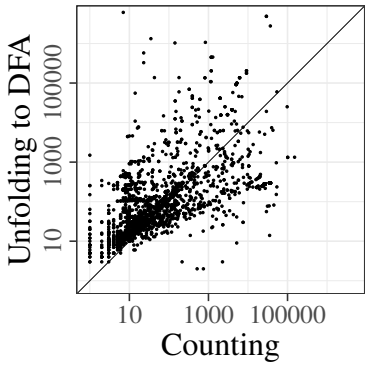


Figure 3.7: Comparison of numbers of transitions (the axes are logarithmic).

representation, in some cases by the factor of hundreds, thousands, or even tens of thousands. Furthermore, the other histogram shows that, in many cases, no blow-up in the number of counters happened; though there are also cases where the number of counters increased by the factor of hundreds.

In terms of numbers of transitions, the methods compare similarly as for numbers of states, as shown in Figure 3.7. We obtained $\mu = 14,282$, $m = 77$, $\sigma = 213,406$ for **DFA** and $\mu = 2,398$, $m = 23$, $\sigma = 8,475$ for **Counting**. (We emphasize the number of states over the number of transitions in our comparisons since the performance and complexity of automata algorithms is usually more sensitive to the number of states, and large numbers of transitions are amenable for efficient symbolic representations [48, 23, 79].)

Benefits of the **Counting** method were the most substantial on the Industrial dataset. For the regex `'.*A[^AB]{0,800}C[D-G]{43,53}DFG[^D-H]'` (which was obtained from the real one, which is confidential, by substituting the used character classes by characters '`A`'–'`H`'), the obtained DFA contains 200,132 states, while the DCA contains only 12 states (and 2 counters), which is 16,667 times less. When minimised, the DFA

still has 65,193 states. There were other regexes where **Counting** achieved a great reduction, in total two regexes had a reduction of over 10,000, three more regexes had a reduction of over 1,000, and 45 more had a reduction of over 100.

Additionally, we also compared our approach against the naive determinisation followed by the standard minimisation. Due to the space restrictions and since minimisation is not relevant to our primary target (preventing failure due to state space explosion during determinisation), we present the results only briefly. Minimisation increased the running times of **DFA** by about one half ($\mu = 150$, $m = 0.35$, $\sigma = 1,582$ for the running times of **DFA** followed by minimisation). The minimal DFAs were on average about ten times smaller than the original DFAs, and about ten times larger than our DCAs ($\mu = 385$, $m = 29$, $\sigma = 4,195$ for the numbers of states of the minimal DFAs).

3.6 Conclusion

We presented a novel procedure for determinising CA based on generalized subset construction. Our algorithm has a potential to produce deterministic CAs exponentially more succinct than the corresponding DFAs. We also developed a simplified and faster version of the general algorithm for the sub-class of so-called monadic CAs (MCAs). We confirmed that monadic regexes present an important subproblem, with over 95 % of regexes in the explored datasets being of this type. The worst-case complexity of the specialised algorithm is only polynomial in the maximum values of repetition bounds (in contrast to the exponential naive construction).

We have implemented the monadic CA determinisation and evaluated it on real-life datasets of regexes with monadic counting. The experiments confirmed that our algorithm produces significantly smaller automata, which are less prone to explode, and that our algorithm, though not optimised, is overall faster than the naive determinisation that unfolds counters. Furthermore, we show large speedups over translation to and subsequent determinisation of symbolic NFAs. Most importantly, we were able to significantly reduce the number of timeouts, thus being able to translate a larger fraction of these datasets into automata.

'Mathematics may be defined as the subject in which we never know what we are talking about, nor whether what we are saying is true.'

Bertrand Russell

4

Regex Matching with Counting-Set Automata

In this chapter, we take the ideas from Chapter 3 a significant step further. In Chapter 3, we proposed a general determinisation of CAs without explicit application to pattern matching which can produce exponentially more succinct automata than the naive determinisation but its worst-case complexity is at best polynomial in the maximum values of counters.

We notice that many counters of the deterministic automata computed by the algorithm in Chapter 5 can be represented as a single data structure. Thus, we introduce a succinct and fast deterministic machine, called the *counting-set automaton (CsA)*, a deterministic finite automaton with a special type of registers that can hold values called *counting sets*, sets of bounded integer values, and support a limited selection of simple set operations. Crucially for the efficiency of our approach, we show that, using a suitable data structure, all the set operations can be implemented to run in *constant time* regardless of the size of the set.

Our compilation from a regex to a CsA proceeds in two steps. First, we convert the regex into a nondeterministic CA. We proposed a novel compilation which generalizes the Antimirov's derivative construction [4]. Its advantage is that it is absent of ϵ-transitions and succinct. The main technical problem we solve in this chapter is a succinct transformation of a (nondeterministic) CA into a deterministic CsA. Our algorithm produces a CsA in *time independent of the repetition bounds* and its simulation is linear in the size of the input text.

We have carried out an extensive experimental evaluation of our algorithm on a large sample of regexes used for pattern matching in various applications. The experiments show that our algorithm, although also limited to a sub-class of regexes, handles over 90 % of regexes with bounded repetition we collected. The obtained data confirm that our CsAs are indeed far smaller and can be constructed faster than corresponding DFAs.

We have implemented a regex matcher called `GadgetCA` based on our generalized Antimirov's algorithm and simulation of CsA obtained by our CA-to-CsA determinisation[1]. The matcher is efficient and applicable to a relatively large class of regexes.

[1] We use a pre-computed deterministic CsA. While on-the-fly determinisation is also possible, it was not

We compared it with several state-of-the-art matchers, namely, `grep` [32], `RE2` [42], `SRM` [82], and `.NET` [65]. Our results show that problematic highly nondeterministic regexes with bounded repetition indeed appear in practice and can also be easily crafted as a ReDoS attack, and that CsAs can efficiently solve most of such problematic cases. For instance, the regex `(_a){64999}_a` from [27] can cause state-of-the-art matchers exceed any reasonable time limit (when searching for the pattern anywhere on the line, with the implicit '`.*`' in front). Already with the repetition bounds lowered to 1,000, the matchers take from 6 to 34 seconds on 500 KB of text, but our algorithm needs only 1 second even with the original bound 64,999.

We summarize the technical contributions of this chapter as follows:

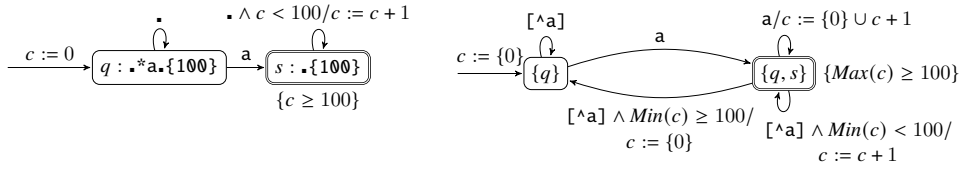
1. A novel Antimirov's style translation from a regex with bounded repetition to a CA.
2. A novel notion of the counting-set automaton, a deterministic machine that allows for succinct representation of repetition constraints and fast matching.
3. CA-to-CsA determinisation that runs in time independent of repetition bounds, the main contribution of this thesis.
4. A regex-matching algorithm interconnecting the above, efficient regardless of repetition bounds especially on regexes that combine bounded repetition with nondeterminism.
5. Implementation and extensive experimental evaluation of the above.

Outline. In Section 4.1, we give a brief overview of our conversion of a regex into a deterministic CsA. Section 4.2 contains preliminaries on effective Boolean algebras, symbolic automata, and counting automata. In Section 4.3, we introduce a generalization of the Antimirov's partial derivative construction to symbolic counting. In Section 4.4, we describe a determinisation algorithm from counting automata to counting-set automata. In Section 4.5, we present the results of our experimentation where we evaluated pattern matching capabilities of our tool `Chipmunk` against other state-of-the-art regex matchers on patterns that use the repetition operator. In Section 4.6, we discuss some important implementation details related to the implementation of algebras, and their role in the CA determinization algorithm. Section 4.7 concludes the chapter and proposes further direction of the research.

4.1 Overview

We give a brief overview of our conversion of a regex with bounded repetition into a deterministic CsA. We use the example regex R = `.*a.{100}` from Example 1.3.1 which represents strings where the symbol `a` appears 100 positions from the end, with the corresponding minimum DFA having 2^{101} states. The conversion proceeds in two steps. First, R is translated into a nondeterministic CA (Figure 4.1a), denoted as $\mathrm{CA}(R)$; second, $\mathrm{CA}(R)$ is converted into a deterministic CsA (Figure 4.1b). The size of both is independent of the repetition bounds (both of the automata will have 2 states only).

needed in our experimentation since we have not witnessed problems with CsA state space explosion.



(a) The CA for the regex `.*a.{`k`}`

(b) The CsA from determinisation of (a)

Figure 4.1: The nondeterministic CA and the deterministic CsA for `.*a.{100}`. The transitions are labeled by their *guard*, which gives the character class (in the standard POSIX regex notation, where, e.g., '`.`' stands for "any character") and possibly restricts counter values, delimited by '/' from the counter *update*. If a counter does not have the update specified, then the transition does not change its value. In Figure 4.1b, the notation $c + 1$ stands for the set of values obtained by incrementing each value in c and then *removing* values larger than the upper bound 100 of the counter. The edges denoting initial states are labelled with *initial values* of the counters. Final states are labelled with an *acceptance condition*, e.g. $\{c \geq 100\}$ in Figure 4.1a.

The formal counter operations OP_c presented later in Section 4.3.3 are in Figure 4.1a shown as follows: the guard of OP_c is shown in conjunction with the character guard α on the left of the '/', the update of OP_c is shown on the right of '/' in the form of an assignment to c, where INCR_c appears as the right value $c + 1$, EXIT as 0, EXIT1 as 1, and NOOP is omitted.

Counting-set data structure. Before looking into the conversion from regular expressions to CsAs it is useful to first understand *why* the resulting CsA can be used efficiently for matching in the first place. The main enabler behind this is the use of our *counting-set* data structure, say c, representing sets $S_c \subseteq \{0, \ldots, \boldsymbol{max}_c\}$ where the upper bound $\boldsymbol{max}_c$ is a fixed positive integer. A *runtime value* of c is a tuple (o, ℓ) where $o \in \mathbb{N}$ is called an *offset* and ℓ is a queue of strictly increasing natural numbers such that $S_c = \{o - n \mid n \in \ell\}$.

The data structure supports *constant-time* implementations of the following operations, assuming constant-time access to the first and the last element of the queue (the queue may be implemented as a doubly-linked list).

- The minimum and the maximum of S_c are obtained as $o - \mathit{last}(\ell)$ and $o - \mathit{first}(\ell)$, respectively.
- Insert 0: if $o - \mathit{last}(\ell) > 0$, then append o at the end of ℓ (similarly for inserting 1).
- Increment all, up to $\boldsymbol{max}_c$: $o := o + 1$; if $o - \mathit{first}(\ell) > \boldsymbol{max}_c$, then remove $\mathit{first}(\ell)$.
- Reset to $\{0\}$: $\ell := 0$; $o := 0$ (similarly for reset to $\{1\}$).

The independence of the running time of these operations of $\boldsymbol{max}_c$ enables our major achievement:

The independence of the running time of pattern matching of the repetition bounds.

Let us now illustrate how this data structure works during matching. We run the CsA in Figure 4.1b., assuming the meaning of the operations provided above, over the sample input word $aa0^{(10)}aab^{(87)}dfa$.

prefix	state	(o, ℓ)	S_c
ϵ	$\{q\}$	$(0, [0])$	$\{0\}$
a	$\{q, s\}$	$(0, [0])$	$\{0\}$
aa	$\{q, s\}$	$(1, [0, 1])$	$\{1, 0\}$
$aa0^{(10)}$	$\{q, s\}$	$(11, [0, 1])$	$\{11, 10\}$
$aa0^{(10)}aa$	$\{q, s\}$	$(13, [0, 1, 12, 13])$	$\{13, 12, 1, 0\}$
$aa0^{(10)}aab^{(87)}$	$\{q, s\}$	$(100, [0, 1, 12, 13])$	$\{100, 99, 88, 87\}$
$aa0^{(10)}aab^{(87)}d$	$\{q, s\}$	$(101, [1, 12, 13])$	$\{100, 89, 88\}$
$aa0^{(10)}aab^{(87)}df$	$\{q, s\}$	$(102, [12, 13])$	$\{90, 89\}$
$aa0^{(10)}aab^{(87)}dfa$	$\{q, s\}$	$(103, [12, 13, 103])$	$\{91, 90, 0\}$

The configurations of the automaton after processing prefixes of the word are shown in the table: the control state, the counting-set run-time configuration (o, ℓ), and the value S_c it represents. The state $\{q, s\}$ fulfills the *accepting condition* after processing the 6th and the 7th prefix since the maximum of S_c at these points is indeed at least 100.

From a nondeterministic CA to a deterministic CsA. The idea of our CA-to-CsA determinisation is best explained by comparison with the naive determinisation of a CA, which would create a DFA by the explicit textbook-style subset construction. The states of the DFA would then be sets of runtime configurations of the CA where each CA-configuration would consist of a control state and a counter valuation. Counter valuations would hence be "unfolded"—they would become an explicit part of the DFA control states—and the succinctness provided by counters would be lost. For instance, the run of the CA in Figure 4.1a on the word $aa0\ldots$ generates "powerstates":

$$\{(q, c=0)\},\ \{(q, c=0), (s, c=0)\},\ \{(q, c=0), (s, c=0), (s, c=1)\},\ \{(q, c=0),$$

$$(s, c=1), (s, c=2)\}, \ldots$$

which are essentially subsets of $\{q, s\} \times \{0, \ldots, 100\}$. In the worst case, the size of the DFA would be exponential in repetition bounds because s can be paired with any subset of $\{0, \ldots, 100\}$ recording possible values of c. In contrast to this, as illustrated above, our CsA represents the counter valuations implicitly: it computes them dynamically on the fly and stores them as *counting sets*—i.e., the valuation of a counter changes from an integer to a *set* of integers. The counter valuations are hence not a part of control states, and their overall number influences neither the size of the CsA nor the time needed to build them.

Figure 4.1b. shows the CsA obtained from determinisation of the CA in Figure 4.1a. The runtime configurations of the CsA reached for the word $aa0$ are:

$$(\{q\}, c=\{0\}),\ \ (\{q, s\}, c=\{0\}),\ \ (\{q, s\}, c=\{0, 1\}),\ \ (\{q, s\}, c=\{1, 2\}).$$

They encode the first three states reached by the sample DFA run above. Namely, the control states are kept in the first component and the counter values are in the second

component, i.e., the set S_c given by the run-time values of the counting-set c. In this encoding, the value of the counting-set is not relevant for the states where the counter is never active (state q in our example). The counter's value in these states is always implicitly 0. In the example, the value S_c of the counting-set therefore only records the values of c at state s and is thus relevant only in the CsA state $\{q, s\}$. We note that for simplicity, we initialise all counting-sets uniformly with $\{0\}$, even if their value is initially irrelevant, as in the case of the CsA state $\{q\}$ in the example.

We note that some DFA powerstates cannot be encoded as CsA configurations due to the involved Cartesian abstraction: essentially, any state in the powerset is paired with any counter value from the counting set. Hence, our approach does not handle the full class of regexes. Fortunately, as our empirical evidence shows, regexes that fall out from the supported class are rare in practice.

From regexes to nondeterministic CAs. To translate a regex into a CA, we propose a generalization of the Antimirov's derivative construction [4] to symbolic counting. In Antimirov's setting, a derivative of a regex R wrt a character class α is a set of regexes that together capture all tails of words in $L(R)$ whose head character is from α. In particular, according to [82], which generalizes [4] to *explicit* counting, the derivatives of the regex $R = \texttt{.*a.\{100\}}$ wrt the classes `a` and `[^a]` are $\{R, \texttt{.\{100\}}\}$ and $\{R\}$, respectively. Further, for $1 \leq k \leq 100$, the derivative of $\texttt{.}\{k\}$ wrt both `a` and `[^a]` is $\{\texttt{.}\{k-1\}\}$. The derivatives become the states of the resulting NFA, with R itself being initial and `.{0}` final, and with α-transitions from each regex to all its α-derivatives (for α being either `a` or `[^a]`). The obtained NFA is already quite large, it has 102 states.

In our new construction, the counting is kept *implicit* using symbolic counters. Instead of modifying the counter bound of the derivative (by, e.g., deriving `.{99}` from `.{100}`), we keep the original bound unchanged and use a counter c to keep track of the difference between the original value and the current value. Our *conditional derivative* operator $\partial_\alpha(\cdot)$ then equips the produced derivatives with *conditional counter updates* to keep the counters up-to-date. For instance, $\partial_{\texttt{a}}(\texttt{.\{100\}})$ returns the same regex `.{100}`, but it is paired with conditional counter updates for c, namely, "if $c < 100$, then increment c; and if $c \geq 100$, then exit the counting loop". The CA we obtain this way is shown in Figure 4.1a, where the first conditional update translates to the self loop on the state `.{100}` and the second to the acceptance condition. The size of the CA does not depend on the repetition bounds.

4.2 Preliminaries

We cast our definitions in the framework of symbolic automata [24]. In this section, we introduce basic notions later used in this chapter.

Effective Boolean algebras. Dealing with large alphabets (such as Unicode) requires succinct representations of automata with many transitions between a pair of states. *Symbolic automata* (systematically studied, e.g., in [24]) use *symbolic transitions* of the form $q \text{-}\{\alpha\}\!\!\mapsto r$ where α is a character class, which represent a set of ordinary explicit transitions $\{q \text{-}\{a\}\!\!\mapsto r \mid a \in \alpha\}$. The notion of determinism on symbolic automata then refers to the represented set of ordinary transitions. Determinisation of

symbolic automata may also be done in a sophisticated way that preserves the succinct representation. Symbolic automata work over alphabets equipped with a so-called *effective Boolean algebra*, which defines the needed interface for handling large sets of labels on automata transitions.

An *effective Boolean algebra* $\mathbb{A}$ has components $(\Sigma, \Psi, [\![\_]\!], \bot, \top, \vee, \wedge, \neg)$ where Σ is a *universe* of underlying domain elements, Ψ is a set of unary *predicates* closed under the Boolean connectives:

- $\vee, \wedge : \Psi \times \Psi \to \Psi$
- and $\neg : \Psi \to \Psi$; and
- $\bot, \top \in \Psi$ are the *false* and *true* predicates.

Values of the algebra are sets of domain elements, and the *denotation function* $[\![\_]\!] : \Psi \to 2^{\Sigma}$ satisfies that:

- $[\![\bot]\!] = \emptyset$,
- $[\![\top]\!] = \Sigma$, and for all $\varphi, \psi \in \Psi$,
- $[\![\varphi \vee \psi]\!] = [\![\varphi]\!] \cup [\![\psi]\!]$,
- $[\![\varphi \wedge \psi]\!] = [\![\varphi]\!] \cap [\![\psi]\!]$, and
- $[\![\neg\varphi]\!] = \Sigma \setminus [\![\varphi]\!]$.

For $\varphi \in \Psi$, we write $\texttt{sat}(\varphi)$ when $[\![\varphi]\!] \neq \emptyset$, and we say that φ is *satisfiable*.

We require that `sat` as well as $\vee$, $\wedge$, and $\neg$ are *computable* as a part of the definition of an effective Boolean algebra. We write $x \models \varphi$ for $x \in [\![\varphi]\!]$ and we use $\mathbb{A}$ as a subscript of a component when it is not clear from the context, e.g., $[\![\_]\!]_{\mathbb{A}} : \Psi_{\mathbb{A}} \to 2^{\Sigma_{\mathbb{A}}}$.

Words and regexes. The basic building blocks of regexes are *predicates* from an effective Boolean algebra *CharClass* of *character classes*. Let $\Sigma = \Sigma_{\mathit{CharClass}}$. The semantics of a regex R is defined as a subset of Σ^* similarly as defined in Section 2.1 with the difference that $L(\alpha) \stackrel{\text{def}}{=} [\![\alpha]\!]$. $R_1 \cdot R_2$ is called a *concat node* and $R_1|R_2$ is called a *choice node*.

We will also need to refer to the number of *character-class leaf nodes* of a regex R, denoted by $\#_\Psi(R)$ and defined as follows:

- $\#_\Psi(\varepsilon) = 0$,
- $\#_\Psi(\alpha) = 1$,
- $\#_\Psi(R_1 \cdot R_2) = \#_\Psi(R_1|R_2) = \#_\Psi(R_1) + \#_\Psi(R_2)$,
- $\#_\Psi(R\{n, m\}) = \#_\Psi(R*) = \#_\Psi(R)$.

Minterms. Let Π_R be the set of all predicates that occur in a regex R, and let $\mathit{Minterms}(R)$ denote the set of *minterms* of Π_R. Intuitively, $\mathit{Minterms}(R)$ is a set of non-overlapping predicates that can be treated as a concrete finite alphabet. Each minterm is essentially a region in the Venn diagram of the predicates in R: it is a satisfiable conjunction $\bigwedge_{\psi \in \Pi_R} \psi'$ where $\psi' \in \{\psi, \neg\psi\}$. For example, if:

$$R = \texttt{[0-z]\{4\}[0-8]\{5\}},$$

then

$$\Pi_R = \{\texttt{[0-8]}, \texttt{[0-z]}\}$$

and

$$\mathit{Minterms}(R) = \{\texttt{[0-8]}, \texttt{[9-z]}, \texttt{[\^0-z]}\}.$$

Formally, if $\alpha \in \mathit{Minterms}(R)$, then $\boldsymbol{Sat}(\alpha)$ and $\forall \psi \in \Pi_R$:

$$[\![\alpha]\!] \cap [\![\psi]\!] \neq \emptyset \Rightarrow [\![\alpha]\!] \subseteq [\![\psi]\!].$$

Note that although the number of minterms of a general set X of predicates may be exponential in $|X|$, it is only linear if X consists of intervals of symbols used in regexes, such as `[a-zA-Z]` or `[^a-zA-Z]` (the former denotes two intervals while the latter their complement, which is equivalent to the union of three intervals). Intervals of numbers generate only a linear number of minterms.

Symbolic automata. We use *symbolic finite automata (FAs)*, whose alphabet is given by an effective Boolean algebra, as a generalization of classical finite automata. Formally, an FA is a tuple $A = (\mathbb{I}, Q, q_0, F, \Delta)$ where $\mathbb{I}$ is an effective Boolean algebra called the input algebra, Q is a finite set of states, $q_0 \in Q$ is the initial state, $F \subseteq Q$ is the set of final states, and $\Delta \subseteq Q \times \Psi_{\mathbb{I}} \times Q$ is a finite set of transitions. A transition $(q, \alpha, r) \in \Delta$ will be also written as $q \text{-}\{\alpha\}\!\!\rightarrow r$.

A *run of A from a state p_0* over a word $a_1 \cdots a_n$ is a sequence of transitions $(p_{i-1} \text{-}\{\alpha_i\}\!\!\rightarrow p_i)_{i=1}^{n}$ with $a_i \in [\![\alpha_i]\!]$; the run is *accepting* if $p_n \in F$. The *language of A from a state q*, denoted $\mathcal{L}_A(q)$, is the set of words over which A has an accepting run from q. The *language of A*, denoted $L(A)$, is $\mathcal{L}_A(q_0)$. A classical finite automaton can be understood as an FA where the basic predicates have singleton set semantics, i.e., when for each concrete letter a there is a predicate α_a such that $[\![\alpha_a]\!] = \{a\}$. A is *deterministic* iff for all $p \in Q$ and all transitions $p \text{-}\{\alpha\}\!\!\rightarrow q$ and $p \text{-}\{\alpha'\}\!\!\rightarrow r$, it holds that if $\alpha \wedge \alpha'$ is satisfiable, then $q = r$.

Counter algebra. A *counter algebra* is an effective Boolean algebra $\mathbb{C}$ associated with a finite set C of counters. The counters play the role of bounded loop variables associated with a *lower bound* $\boldsymbol{min}_c \geq 0$ and an *upper bound* $\boldsymbol{max}_c > 0$ such that $\boldsymbol{min}_c \leq \boldsymbol{max}_c$. $\Sigma_{\mathbb{C}}$ is the set of interpretations $\mathfrak{m} : C \to \mathbb{N}$, called *counter memories* such that $0 \leq \mathfrak{m}(c) \leq \boldsymbol{max}_c$ for all $c \in C$. $\Psi_{\mathbb{C}}$ contains Boolean combinations of *basic* predicates $\textsc{canExit}_c$ and $\textsc{canIncr}_c$, for $c \in C$, whose semantics is given by:

$$\mathfrak{m} \models \textsc{canExit}_c \iff \mathfrak{m}(c) \geq \boldsymbol{min}_c, \quad \mathfrak{m} \models \textsc{canIncr}_c \iff \mathfrak{m}(c) < \boldsymbol{max}_c.$$

Counting automata. In this chapter, we use a slightly different (more straightforward) notation for counting automaton than in Chapter 3.

A *counting automaton (CA)* is a tuple $A = (\mathbb{I}, C, Q, q_0, F, \Delta)$ where $\mathbb{I}$ is an effective Boolean algebra called the *input algebra*, C is a finite set of *counters* with an associated counter algebra $\mathbb{C}$, Q is a finite set of *states*, $q_0 \in Q$ is the *initial state*, $F : Q \to \Psi_{\mathbb{C}}$ is the *final-state condition*, and $\Delta \subseteq Q \times \Psi_{\mathbb{I}} \times (C \to O) \times Q$ is the (finite) *transition relation*, where $O = \{\textsc{exit}, \textsc{incr}, \textsc{exit1}, \textsc{noop}\}$ is the set of *counter operations*. The component f of a transition $p \text{-}\{\alpha, f\}\!\!\to q \in \Delta$ is its *(counter) operator*. We often view f as a set of *indexed operations* $\textsc{op}_c$ to denote the operation assigned to the counter c, $f(c) = \textsc{op}$.

Semantics of CAs. The semantics of the CA A is defined through its *configuration automaton* $FA(A)$, an FA whose states are A's *configurations*, i.e., pairs $(q, \mathfrak{m}) \in Q \times \Sigma_{\mathbb{C}}$ consisting of a state q and a counter memory $\mathfrak{m}$. To define $FA(A)$, we must first define the semantics of counter operators f, which occur on transitions. For this, we associate with each (indexed) operation $\textsc{op}_c$ a counter guard $grd(\textsc{op}_c)$ and a counter update $upd(\textsc{op})$ as shown on the right. Intuitively, the operation NOOP does not modify the counter's value and is always enabled. The operation INCR increments the counter and is enabled if the counter has not yet reached its upper bound. The operation EXIT resets the counter to 0 on exit from the counting loop and is enabled when the counter reaches its lower bound. The operation EXIT1 executes EXIT followed by INCR. The *guard* of a counter operator $f : C \to O$ is then a predicate $\varphi_f \in \Psi_{\mathbb{C}}$ over counter memories, and its *update* $\mathbf{f} : \Sigma_{\mathbb{C}} \cup \{\bot\} \to \Sigma_{\mathbb{C}} \cup \{\bot\}$ is a counter-memory transformer:

$$\begin{array}{ll} grd(\textsc{noop}_c) \stackrel{\text{def}}{=} \top_{\mathbb{C}} & upd(\textsc{noop}) \stackrel{\text{def}}{=} \boldsymbol{\lambda} n.n \\ grd(\textsc{incr}_c) \stackrel{\text{def}}{=} \textsc{canIncr}_c & upd(\textsc{incr}) \stackrel{\text{def}}{=} \boldsymbol{\lambda} n.n+1 \\ grd(\textsc{exit}_c) \stackrel{\text{def}}{=} \textsc{canExit}_c & upd(\textsc{exit}) \stackrel{\text{def}}{=} \boldsymbol{\lambda} n.0 \\ grd(\textsc{exit1}_c) \stackrel{\text{def}}{=} \textsc{canExit}_c & upd(\textsc{exit1}) \stackrel{\text{def}}{=} \boldsymbol{\lambda} n.1 \end{array}$$

$$\varphi_f \stackrel{\text{def}}{=} \bigwedge_{\textsc{op}_c \in f} grd(\textsc{op}_c) \tag{4.1}$$

$$\mathbf{f}(\mathfrak{m}) \stackrel{\text{def}}{=} \begin{cases} \boldsymbol{\lambda} c.upd(f(c))(\mathfrak{m}(c)) & \text{if } \mathfrak{m} \models \varphi_f \\ \bot & \text{otherwise} \end{cases} \tag{4.2}$$

Intuitively, $\mathbf{f}$ updates all counters in a counter memory $\mathfrak{m}$ by their corresponding operations if $\mathfrak{m}$ satisfies the guard, otherwise the result is $\bot$.

Let $p \text{-}\{\alpha, f\}\!\!\to q \in \Delta$ and $c \in C$. We use φ_f^c to denote a guard for a counter c inside φ_f. Let $k \in N$. We use $k \models \varphi_f^c$ to denote that k satisfies the guard for a counter c.

We now define the *configuration automaton* of A, denoted as $FA(A)$, which defines the language semantics of the CA A. The states of $FA(A)$ are the configurations of A (there are finitely many of them), and the initial state of $FA(A)$ is the *initial configuration* $(q_0, \{c \mapsto 0 \mid c \in C\})$ of A. A state $(p, \mathfrak{m})$ of $FA(A)$ is *final* iff $\mathfrak{m} \models F(p)$. The transition relation $\Delta_{FA(A)}$ of $FA(A)$ is defined as:

$$\Delta_{FA(A)} = \{(p, \mathfrak{m}) \text{-}\{\alpha\}\!\!\to (q, \mathbf{f}(\mathfrak{m})) \mid p \text{-}\{\alpha, f\}\!\!\to q \in \Delta, \mathfrak{m} \models \varphi_f\}.$$

Deterministic and simple CAs. A is *deterministic* iff the following holds for every state $p \in Q$ and every two transitions $p \text{-}\{\alpha_1,f_1\}\!\mapsto q_1$, $p \text{-}\{\alpha_2,f_2\}\!\mapsto q_2 \in \Delta$: if both $\alpha_1 \wedge \alpha_2$ and $\varphi_{f_1} \wedge \varphi_{f_2}$ are satisfiable, then $f_1 = f_2$ and $q_1 = q_2$. It follows from the definitions that, if A is deterministic, then $FA(A)$ is deterministic too. A is *simple* if for any two transitions $q \text{-}\{\alpha,f\}\!\mapsto r$ and $q' \text{-}\{\alpha',f'\}\!\mapsto r'$, either $\alpha = \alpha'$ or $[\![\alpha]\!] \cap [\![\alpha']\!] = \emptyset$. That is, different character guards do not overlap and can be mostly treated as plain symbols. We also require that all guards are satisfiable. CAs constructed from regexes by the algorithm in Section 4.3 will be simple.

Example 4.2.1. Figure 4.1a shows a CA in an intuitive notation, with the initial state q and final conditions $F(q) = \bot$, $F(s) = \textsc{exit}_c$, where $\boldsymbol{min}_c = \boldsymbol{max}_c = 100$. The same notation is used in Figure 4.2. Figure 4.3a shows a CA in a notation following the formal development more closely. □

4.3 From Regexes to CAs via Conditional Partial Derivatives

We introduce a generalization of the Antimirov's partial derivative construction [4] to *symbolic* counting, which allows one to replace a verbose NFA by a succinct CA. The difference between the older variant of [4] with *explicit* counting [82] and our new version was already illustrated in Section 4.1. To recall it briefly using the example of the regex `_{100}`: from 100 partial derivatives $\partial_{\_}(\texttt{\_\{i\}}) = \texttt{\_\{i-1\}}$, $1 \leq i \leq 100$, and an NFA with 100 states and transitions `_{i}`$\text{-}\{\_\}\!\mapsto$`_{i-1}`, the new construction will take us to the single derivative $\partial_{\_}(\texttt{\_\{100\}}) = \{\texttt{\_\{100\}}\}$ associated with a conditional counter update which induce an NFA with a single state and the transition `_{100}`$\text{-}\{\alpha,\textsc{incr}_c\}\!\mapsto$`_{100}`.

We apply the construction on regexes that are normalized using the below rules where $X \rightsquigarrow Y$ denotes that X is rewritten to Y:

- All nested concat nodes are rewritten to the flattened right-associative *list form*, which is always maintained throughout the construction, using the rules: $(X \cdot Y) \cdot Z \rightsquigarrow X \cdot (Y \cdot Z)$, $\varepsilon \cdot Z \rightsquigarrow Z$, and $Z \cdot \varepsilon \rightsquigarrow Z$.
- If S is *nullable*, then $S\{\ell,k\} \rightsquigarrow S\{0,k\}$. Moreover, in the nullable context $S\{0,k\}$, S can be considered as if it was not nullable.

Observe that the normalization does not increase the size of the regex (it may decrease the size).

Let R be a fixed normalized regex. A subexpression of R that is of the form $X = S\{\ell,k\}$ is called a *counting loop*. We consider each counting loop to represent a *counter* whose name is the counting loop itself and whose *upper bound* is $\boldsymbol{max}_X = k$ and *lower bound* is $\boldsymbol{min}_X = \ell$. For example, `(_{9})*` contains the counter $X =$ `_{9}` with $\boldsymbol{min}_X = \boldsymbol{max}_X = 9$. In the following, let C stand for the set of all counters that occur in R, also denoted by $Counters(R)$.

We use the convention that the juxtaposition XY of normalized regexes X and Y is again a normalized regex of the equivalent concat node $X \cdot Y$: e.g., if $X = a \cdot b$ and $Y = (a \cdot b)*$, then $XY = a \cdot (b \cdot (a \cdot b)*)$. Observe in particular that $X\varepsilon = X$. In other words, we treat concatenated elements as sequences, and a singleton sequence equals to the element itself.

Our construction will work over the set $\Sigma = Minterms(R)$ of minterms of R and produce simple CA that use minterms of Σ on transitions.

4.3.1 Parametric Languages

We define the language of a normalized regex starting with a counting loop relative to a counter value. For that, we lift the definition of languages to be parametric in counter memories $\mathfrak{m}$, but regexes other than the above are treated as before with the memory $\mathfrak{m}$ passed through unchanged.

$$\mathbf{L}_{\mathfrak{m}}(\varepsilon) \stackrel{\text{def}}{=} \{\epsilon\} \tag{4.3}$$

$$\mathbf{L}_{\mathfrak{m}}(\psi Z) \stackrel{\text{def}}{=} [[\psi]] \cdot \mathbf{L}_{\mathfrak{m}}(Z) \tag{4.4}$$

$$\mathbf{L}_{\mathfrak{m}}((W|Y)Z) \stackrel{\text{def}}{=} \mathbf{L}_{\mathfrak{m}}(WZ) \cup \mathbf{L}_{\mathfrak{m}}(YZ) \tag{4.5}$$

$$\mathbf{L}_{\mathfrak{m}}(S*Z) \stackrel{\text{def}}{=} \mathbf{L}_{\mathfrak{m}}(S)^* \cdot \mathbf{L}_{\mathfrak{m}}(Z) \tag{4.6}$$

$$\mathbf{L}_{\mathfrak{m}}(S\{\ell,k\}Z) \stackrel{\text{def}}{=} \mathbf{L}_{\mathfrak{m}}(S) \cdot \mathbf{L}_{\text{INCR}_{S\{\ell,k\}}(\mathfrak{m})}(S\{\ell,k\}Z) \cup \mathbf{L}_{\text{EXIT}_{S\{\ell,k\}}(\mathfrak{m})}(Z) \tag{4.7}$$

$$\mathbf{L}_{\perp}(X) \stackrel{\text{def}}{=} \emptyset \quad \text{(for all } X\text{)} \tag{4.8}$$

Recall that if f is a counter operator and $\mathfrak{m}$ a counter memory, then $\mathbf{f}(\mathfrak{m})$ denotes the appropriately updated memory where $\mathbf{f}(\mathfrak{m}) = \perp$ when f is not enabled in $\mathfrak{m}$. Below, if there is a single counter $c \in C$ such that $f(c) \neq \text{NOOP}$, we sometimes identify f with OP_c and use $\text{OP}_c(\mathfrak{m})$ to represent the updated memory $\mathbf{f}(\mathfrak{m})$. Specifically, INCR_X (if enabled) increments the counter value of X by 1, and EXIT_X (if enabled) resets the counter value of X to 0. Let $\mathfrak{m}$ be a counter memory. Then Cases (4.3)–(4.8) define the *parametric languages* of regexes. The intuition behind Case (4.6) is that all counters that may be present in S are inactive on the level of $S*$. Note also that Case (4.7) is well-defined since, for $X = S\{\ell,k\}$ and $\mathfrak{m}' = \text{INCR}_X(\mathfrak{m})$, $k - \mathfrak{m}'(X) < k - \mathfrak{m}(X)$ if $\mathfrak{m}(X) < k$, and $\mathfrak{m}' = \perp$ if $\mathfrak{m}(X) = k$. Let $\mathbf{0} \stackrel{\text{def}}{=} \lambda c.0$ denote the initial memory that maps all counters to 0.

We need the following additional notions in order to reason about correctness of the construction of CAs from regexes via conditional partial derivatives stated in Theorem 4. Let R be a normalized regex. A counter X *is visible in* R, denoted $X \in \mathit{Visible}(R)$, if either $R = YZ$ and $X = Y$, or else if X does not occur in Y and X is visible in Z, i.e., $X \in \mathit{Visible}(Z) \setminus \mathit{Counters}(Y)$. In other words,

$$\mathit{Visible}(R) = \begin{cases} \emptyset, & \text{if } R = \varepsilon; \\ \{S\{\ell,k\}\} \cup (\mathit{Visible}(Z) \setminus \mathit{Counters}(S)), & \text{else if } R = S\{\ell,k\}Z; \\ \mathit{Visible}(Z) \setminus \mathit{Counters}(Y), & \text{otherwise, where } R = YZ. \end{cases}$$

Let $\mathit{Hidden}(R) \stackrel{\text{def}}{=} \mathit{Counters}(R) \setminus \mathit{Visible}(R)$. A counter memory $\mathfrak{m}$ is *valid for* R if $\mathfrak{m}(X) = 0$ for all $X \in \mathit{Hidden}(R)$, otherwise $\mathfrak{m}$ is *invalid* for R. Intuitively, every hidden counter in R must have the initial value 0 in any valid memory. We only consider counter memories $\mathfrak{m}$ that are valid for R in the context of $\mathbf{L}_{\mathfrak{m}}(R)$.

Example 4.3.1. Let $X = \texttt{a\{3\}}$. Then X is visible in $X \cdot X*$ but hidden in $X* \cdot X$. A counter memory $\mathfrak{m}$ such that $\mathfrak{m}(X) = 2$ is valid for $X \cdot X*$ but invalid for $X* \cdot X$. □

Example 4.3.2. Let X be the regex `(a(bc){7}d){8}`. Then X and $Y = \texttt{(bc)\{7\}}$ are both counters in X but only X is visible in X. Now consider the regex $Y\texttt{d}X$, or more precisely $Y \cdot (\texttt{d} \cdot X)$ to emphasize the normalized form. In this case $\mathit{Visible}(Y\texttt{d}X) = \{X, Y\}$.

the visible counters of $\mathtt{c}Y\mathtt{d}X$ are $\{X, Y\}$. In fact, as shown below, $\{\langle \textsc{incr}_X, Y\mathtt{d}X\rangle\}$ is the (conditional) $\mathtt{a}$-derivative of X and $\{\langle \textsc{incr}_Y, \mathtt{c}Y\mathtt{d}X\rangle\}$ is the $\mathtt{b}$-derivative of $Y\mathtt{d}X$. These are in fact the only possible regexes that arise here through derivation starting with X. the $\mathtt{c}$-derivative of $\mathtt{c}Y\mathtt{d}X$ is $\{\langle \lambda x.x, Y\mathtt{d}X\rangle\}$, and the $\mathtt{d}$-derivative of $Y\mathtt{d}X$ is $\{\langle \textsc{exit}_Y, X\rangle\}$. All other derivatives are empty. □

We use the following lemma in the correctness theorem of partial derivatives. If E is a set of counters, then $\rho(E)$ resets the values of all counters in E to 0. Observe that $\rho(\{c\})$ is in general different from $\textsc{exit}_c$ because $\textsc{exit}_c(\mathfrak{m}) = \bot$ when $\mathfrak{m} \not\models \textsc{canExit}_c$.

In the proof of the lemma (and also multiple further proofs), we will need the notion of the *size* of a regex R, denoted by $\#(R)$, which corresponds to the number of nodes in the abstract syntax tree of R (apart from the case when R is ϵ) and which we define as follows:

$$\#(\varepsilon) = 0 \quad \#(\alpha) = 1 \quad \#(R_1 \cdot R_2) = \#(R_1) + \#(R_2) + 1$$
$$\#(R_1|R_2) = \#(R_1) + \#(R_2) + 1 \quad \#(R\{n,m\}) = \#(R) + 1 \quad \#(R*) = \#(R) + 1$$

Lemma 4.1. *If X and Y are normalized regexes and $\mathfrak{m}$ is valid for XY then*

$$\mathbf{L}_{\mathfrak{m}}(XY) = \mathbf{L}_{\mathfrak{m}}(X) \cdot \mathbf{L}_{\rho(\mathit{Visible}(X))(\mathfrak{m})}(Y).$$

Proof. By induction over the pair $(\#(X), n)$ where $n = \boldsymbol{max}_c - \mathfrak{m}(c)$ if X starts with a counter c or else $n = 0$. Let $(m', n') < (m, n)$ iff either $m' < m$ or else $m' = m$ and $n' < n$.

Base case $X = \varepsilon$ Then $\rho(\mathit{Visible}(\epsilon))(\mathfrak{m}) = \mathfrak{m}$ and $\mathbf{L}_{\mathfrak{m}}(\varepsilon) = \{\epsilon\}$.

Induction case $X = S\{\ell, k\}Z$ Let $c = S\{\ell, k\}$, $\mathfrak{m}_0 = \textsc{exit}_c(\mathfrak{m})$ and $\mathfrak{m}_1 = \textsc{incr}_c(\mathfrak{m})$.

- Case $\mathfrak{m}(c) = \boldsymbol{max}_c$. Then $\mathfrak{m}_1 = \bot$ and $\mathfrak{m}_0$ is valid for ZY because $\mathfrak{m}$ is valid for XY and $\mathfrak{m}_0(c) = 0$. (Observe that if $d \in \mathit{Hidden}(ZY)$ then either $\mathfrak{m}_0(d) = 0$ if $d = c$ or else $\mathfrak{m}(d) = 0$ because then $d \in \mathit{Hidden}(XY)$.) It follows that

$$\begin{aligned}
\mathbf{L}_{\mathfrak{m}}(XY) &\overset{(\mathfrak{m}_1=\bot)}{=} \mathbf{L}_{\mathfrak{m}_0}(ZY) \\
&\overset{\text{(IH)}}{=} \mathbf{L}_{\mathfrak{m}_0}(Z) \cdot \mathbf{L}_{\rho(\mathit{Visible}(Z))(\mathfrak{m}_0)}(Y) \\
&\overset{(\mathfrak{m}_1=\bot)}{=} \mathbf{L}_{\mathfrak{m}}(X) \cdot \mathbf{L}_{\rho(\mathit{Visible}(Z))(\textsc{exit}_c(\mathfrak{m}))}(Y) \\
&= \mathbf{L}_{\mathfrak{m}}(X) \cdot \mathbf{L}_{\rho(\mathit{Visible}(X))(\mathfrak{m})}(Y)
\end{aligned}$$

  where the last equality holds because $\mathit{Visible}(X) = \{c\} \cup (\mathit{Visible}(Z) \setminus \mathit{Counters}(S))$ and $\mathfrak{m}(d) = 0$ for all $d \in \mathit{Counters}(S)$ because $\mathfrak{m}$ is valid for X. Hence $\rho(\mathit{Visible}(Z))(\textsc{exit}_c(\mathfrak{m})) = \rho(\mathit{Visible}(X))(\mathfrak{m})$.

- Case $\mathfrak{m}(c) < \boldsymbol{min}_c$: Then $\mathfrak{m}_0 = \bot$. Here $\mathfrak{m}_1$ is valid for XY because $\mathfrak{m}$ is valid and $c \in \mathit{Visible}(XY)$. Also, the (IH) applies because $k - \mathfrak{m}_1(c) < k - \mathfrak{m}(c)$.

$$\begin{aligned}
\mathbf{L}_{\mathfrak{m}}(XY) &\overset{(\mathfrak{m}_0=\bot)}{=} \mathbf{L}_{\mathfrak{m}}(S) \cdot \mathbf{L}_{\mathfrak{m}_1}(XY) \\
&\overset{\text{(IH)}}{=} \mathbf{L}_{\mathfrak{m}}(S) \cdot \mathbf{L}_{\mathfrak{m}_1}(X) \cdot \mathbf{L}_{\rho(\mathit{Visible}(X))(\mathfrak{m}_1)}(Y) \\
&\overset{(c \in \mathit{Visible}(X))}{=} \mathbf{L}_{\mathfrak{m}}(S) \cdot \mathbf{L}_{\mathfrak{m}_1}(X) \cdot \mathbf{L}_{\rho(\mathit{Visible}(X))(\mathfrak{m})}(Y) \\
&\overset{(\mathfrak{m}_0=\bot)}{=} \mathbf{L}_{\mathfrak{m}}(X) \cdot \mathbf{L}_{\rho(\mathit{Visible}(X))(\mathfrak{m})}(Y)
\end{aligned}$$

- Case $\boldsymbol{min}_c \leq \mathfrak{m}(c) < \boldsymbol{max}_c$. Then $\mathfrak{m}_0 \neq \bot$ and $\mathfrak{m}_1 \neq \bot$. This case is a combination of the above two cases where the IH is applied twice under similar conditions.

$$\begin{array}{rcl}
\mathbf{L}_{\mathfrak{m}}(XY) & = & \mathbf{L}_{\mathfrak{m}}(S) \cdot \mathbf{L}_{\mathfrak{m}_1}(XY) \cup \mathbf{L}_{\mathfrak{m}_0}(ZY) \\
& \overset{(2\times\text{IH})}{=} & \mathbf{L}_{\mathfrak{m}}(S) \cdot \mathbf{L}_{\mathfrak{m}_1}(X) \cdot \mathbf{L}_{\rho(\mathit{Visible}(X))(\mathfrak{m}_1)}(Y) \cup \mathbf{L}_{\mathfrak{m}_0}(Z) \cdot \\
& & \mathbf{L}_{\rho(\mathit{Visible}(Z))(\mathfrak{m}_0)}(Y) \\
& = & \mathbf{L}_{\mathfrak{m}}(S) \cdot \mathbf{L}_{\mathfrak{m}_1}(X) \cdot \mathbf{L}_{\rho(\mathit{Visible}(X))(\mathfrak{m})}(Y) \cup \mathbf{L}_{\mathfrak{m}_0}(Z) \cdot \\
& & \mathbf{L}_{\rho(\mathit{Visible}(X))(\mathfrak{m})}(Y) \\
& = & \big(\mathbf{L}_{\mathfrak{m}}(S) \cdot \mathbf{L}_{\mathfrak{m}_1}(X) \cup \mathbf{L}_{\mathfrak{m}_0}(Z)\big) \cdot \mathbf{L}_{\rho(\mathit{Visible}(X))(\mathfrak{m})}(Y) \\
& = & \mathbf{L}_{\mathfrak{m}}(X) \cdot \mathbf{L}_{\rho(\mathit{Visible}(X))(\mathfrak{m})}(Y)
\end{array}$$

Induction case $X = \psi Z$ Trivially $\mathit{Visible}(X) = \mathit{Visible}(Z)$.

$$\begin{array}{rcl}
\mathbf{L}_{\mathfrak{m}}(XY) & = & [\![\psi]\!] \cdot \mathbf{L}_{\mathfrak{m}}(ZY) \\
& \overset{(\text{IH})}{=} & [\![\psi]\!] \cdot \mathbf{L}_{\mathfrak{m}}(Z) \cdot \mathbf{L}_{\rho(\mathit{Visible}(Z))(\mathfrak{m})}(Y) \\
& = & \mathbf{L}_{\mathfrak{m}}(X) \cdot \mathbf{L}_{\rho(\mathit{Visible}(X))(\mathfrak{m})}(Y)
\end{array}$$

Induction case $X = (A|B)Z$ Let $X_1 = AZ$ and $X_2 = BZ$. Here $\mathit{Visible}(X) = \mathit{Visible}(Z) \setminus \mathit{Counters}(A|B)$. Thus, for all $c \in \mathit{Counters}(A|B)$, $\mathfrak{m}(c) = 0$ because $\mathfrak{m}$ is valid for X. Therefore, if $c \in \mathit{Visible}(X_i)$ then either $c \in \mathit{Visible}(X)$ or else $c \in \mathit{Counters}(A|B)$ and $\mathfrak{m}(c) = 0$. Hence $\rho(\mathit{Visible}(X_i))(\mathfrak{m}) = \rho(\mathit{Visible}(X))(\mathfrak{m})$.

$$\begin{array}{rcl}
\mathbf{L}_{\mathfrak{m}}(XY) & = & \mathbf{L}_{\mathfrak{m}}(X_1Y) \cup \mathbf{L}_{\mathfrak{m}}(X_2Y) \\
& \overset{(2\times\text{IH})}{=} & \mathbf{L}_{\mathfrak{m}}(X_1) \cdot \mathbf{L}_{\rho(\mathit{Visible}(X_1))(\mathfrak{m})}(Y) \cup \mathbf{L}_{\mathfrak{m}}(X_2) \cdot \mathbf{L}_{\rho(\mathit{Visible}(X_2))(\mathfrak{m})}(Y) \\
& = & \mathbf{L}_{\mathfrak{m}}(X_1) \cdot \mathbf{L}_{\rho(\mathit{Visible}(X))(\mathfrak{m})}(Y) \cup \mathbf{L}_{\mathfrak{m}}(X_2) \cdot \mathbf{L}_{\rho(\mathit{Visible}(X))(\mathfrak{m})}(Y) \\
& = & (\mathbf{L}_{\mathfrak{m}}(X_1) \cup \mathbf{L}_{\mathfrak{m}}(X_2)) \cdot \mathbf{L}_{\rho(\mathit{Visible}(X))(\mathfrak{m})}(Y) \\
& = & \mathbf{L}_{\mathfrak{m}}(X) \cdot \mathbf{L}_{\rho(\mathit{Visible}(X))(\mathfrak{m})}(Y)
\end{array}$$

Induction case $X = S{*}Z$ Clearly $\mathfrak{m}$ is valid for Z because it is valid for X. Here $\mathit{Visible}(X) = \mathit{Visible}(Z) \setminus \mathit{Counters}(S)$. So if $c \in \mathit{Visible}(Z)$ then either $c \in \mathit{Visible}(X)$ or else $\mathfrak{m}(c) = 0$. Thus $\rho(\mathit{Visible}(Z))(\mathfrak{m}) = \rho(\mathit{Visible}(X))(\mathfrak{m})$.

$$\begin{array}{rcl}
\mathbf{L}_{\mathfrak{m}}(XY) & = & \mathbf{L}_{\mathfrak{m}}(S)^* \cdot \mathbf{L}_{\mathfrak{m}}(ZY) \\
& \overset{(\text{IH})}{=} & \mathbf{L}_{\mathfrak{m}}(S)^* \cdot \mathbf{L}_{\mathfrak{m}}(Z) \cdot \mathbf{L}_{\rho(\mathit{Visible}(Z))(\mathfrak{m})}(Y) \\
& = & \mathbf{L}_{\mathfrak{m}}(S)^* \cdot \mathbf{L}_{\mathfrak{m}}(Z) \cdot \mathbf{L}_{\rho(\mathit{Visible}(X))(\mathfrak{m})}(Y) \\
& = & \mathbf{L}_{\mathfrak{m}}(X) \cdot \mathbf{L}_{\rho(\mathit{Visible}(X))(\mathfrak{m})}(Y)
\end{array}$$

the statement follows by the induction principle. □

The following theorem, relates $\mathbf{L}_{\mathfrak{m}}(R)$ with the non-parametric definition of regular languages.

Theorem 3. *Let R be a normalized regex. Then $\mathbf{L_0}(R) = L(R)$.*

We first prove Lemma 4.2 that states a property used in the proof of Theorem 3. Note that $\mathbf{0}$ is trivially valid for any regex R. Let $S\{0,0\} \stackrel{\text{def}}{=} \varepsilon$ and let $m \ominus n \stackrel{\text{def}}{=} \boldsymbol{max}(m-n, 0)$.

Lemma 4.2. *Let $X = S\{\ell, k\}$ be a normalized counting loop and let $\mathfrak{m}$ be valid for X. Then*

$$\mathbf{L}_{\mathfrak{m}}(X) = \bigcup_{i=\ell\ominus\mathfrak{m}(X)}^{k-\mathfrak{m}(X)} \mathbf{L_0}(S)^{(i)}.$$

Proof. By induction over $k - n$ where $n = \mathfrak{m}(X)$. Let $\mathfrak{m}_1 = \textsc{incr}_X(\mathfrak{m})$. Let $L = \mathbf{L_0}(S)$.

Base case $n = k$ Then $\mathfrak{m}_1 = \bot$ and so $\mathbf{L}_{\mathfrak{m}}(X) = \mathbf{L}_{\textsc{exit}_X(\mathfrak{m})}(\varepsilon) = \{\epsilon\}$ because $\ell \leq k$ and so $\mathfrak{m} \models \textsc{canExit}_X$ and, by definition, $L^{(0)} = \{\epsilon\}$ for any L.

Induction case $n < k$ Here $m_1 \neq \bot$ and $\mathbf{L}_{\mathfrak{m}}(S) = \mathbf{L_0}(S) = L$ because $\mathit{Visible}(X) = \{X\}$.

$$\begin{array}{rcl}
\mathbf{L}_{\mathfrak{m}}(X) & = & \mathbf{L}_{\mathfrak{m}}(S) \cdot \mathbf{L}_{\mathfrak{m}_1}(X) \cup \mathbf{L}_{\textsc{exit}_X(\mathfrak{m})}(\varepsilon) \\
& = & L \cdot \mathbf{L}_{\mathfrak{m}_1}(X) \cup \{\epsilon \mid \ell \leq n\} \\
& \stackrel{\text{(IH)}}{=} & L \cdot \Big(\bigcup_{i=\ell\ominus(n+1)}^{k-(n+1)} L^{(i)} \Big) \cup \{\epsilon \mid \ell \leq n\} \\
& = & \Big(\bigcup_{i=(\ell\ominus(n+1))+1}^{k-n} L^{(i)} \Big) \cup \{\epsilon \mid \ell \leq n\} \quad = \quad \bigcup_{i=\ell\ominus n}^{k-n} L^{(i)}
\end{array}$$

The last two equalities use standard rules of set theory and theory of sequences. □

We can now prove Theorem 3.

Proof of Theorem 3. By induction over $\#(R)$. the base case $R = \varepsilon$ is trivial. the main induction case is $R = S\{\ell, k\}Z$. Then

$$\begin{array}{rcl}
\mathbf{L_0}(R) & \stackrel{\text{(Lemma 4.1)}}{=} & \mathbf{L_0}(S\{\ell, k\}) \cdot \mathbf{L_0}(Z) \\
& \stackrel{\text{(Lemma 4.2)}}{=} & \Big(\bigcup_{i=\ell}^{k} \mathbf{L_0}(S)^{(i)} \Big) \cdot \mathbf{L_0}(Z) \\
& \stackrel{\text{(2×IH)}}{=} & \Big(\bigcup_{i=\ell}^{k} L(S)^{(i)} \Big) \cdot L(Z) \\
& = & L(S\{\ell, k\}) \cdot L(Z) \\
& = & L(R)
\end{array}$$

the remaining cases follow by induction. □

4.3.2 Conditional Derivation

In Section 2.5, we present a general definition for derivative construction. Now we will introduce our conditional derivative construction for regexes with a counter operator formally. A *partial conditional derivative* is a pair $\langle f, X\rangle$ where f is a counter operator and X a normalized regex. Given a counter memory $\mathfrak{m}$, we let $\langle f, X\rangle$ define the language $\mathbf{L}_{\mathfrak{m}}(\langle f, X\rangle) \stackrel{\text{def}}{=} \mathbf{L}_{\mathbf{f}(\mathfrak{m})}(X)$. In other words, f is first applied to the counter memory $\mathfrak{m}$ and then the regex is evaluated in the updated memory. If f is not enabled in $\mathfrak{m}$, then the denoted language is empty.

A *conditional derivative* is a finite set of partial conditional derivatives. The language defined by a conditional derivative D in a counter memory $\mathfrak{m}$ is defined as the union of the languages of the partial conditional derivatives in D: $\mathbf{L}_{\mathfrak{m}}(D) \stackrel{\text{def}}{=} \bigcup_{d\in D} \mathbf{L}_{\mathfrak{m}}(d)$.

To define how conditional derivatives of a given regex looks like, we need a notion of a *sequential composition* of conditional derivatives $D \otimes E \stackrel{\text{def}}{=} \{\langle f; g, X{\cdot}Y\rangle \mid \langle f, X\rangle \in D, \langle g, Y\rangle \in E, f; g \neq \bot\}$ where $f; g \neq \bot$ is the composed counter operator such that $\mathbf{f}; \mathbf{g}(\mathfrak{m}) = \mathbf{g}(\mathbf{f}(\mathfrak{m}))$. The case when $f; g = \bot$ is discussed later on.

$$
\begin{aligned}
\partial_\alpha(\varepsilon) &\stackrel{\text{def}}{=} \emptyset \\
\partial_\alpha(\psi Z) &\stackrel{\text{def}}{=} \begin{cases} \{\langle \textbf{ID}, Z\rangle\} & \text{if } \alpha \wedge \psi \text{ is satisfiable} \\ \emptyset & \text{otherwise} \end{cases} \\
\partial_\alpha((W|Y)Z) &\stackrel{\text{def}}{=} \partial_\alpha(WZ) \cup \partial_\alpha(YZ) \\
\partial_\alpha(S{*}Z) &\stackrel{\text{def}}{=} \partial_\alpha(S) \otimes \{\langle \textbf{ID}, S{*}Z\rangle\} \cup \partial_\alpha(Z) \\
\partial_\alpha(XZ) &\stackrel{\text{def}}{=} \partial_\alpha(S) \otimes \{\langle \textsc{incr}_X, XZ\rangle\} \ \cup \\
&\qquad \{\langle \textsc{exit}_X, \varepsilon\rangle\} \otimes \partial_\alpha(Z)
\end{aligned}
$$

Conditional derivatives of a normalized regex are defined as shown on the right assuming that concatenations $X{\cdot}Y$ are normalized to the list form explained above, $\alpha \in \Sigma$, $\textbf{ID}$ denotes the identity function $\lambda x.x$, and $X = S\{\ell, k\}$ is a counting loop. Observe that, in $\partial_\alpha(S) \otimes \{\langle \textsc{incr}_X, XZ\rangle\}$, the operation $\textsc{incr}_X$ gets composed with $\textsc{noop}_X$, yielding $\textsc{incr}_X$ again, because $S\{\ell, k\}$ cannot occur in S. It is possible that in $\{\langle \textsc{exit}_X, \varepsilon\rangle\} \otimes \partial_\alpha(Z)$, X is in scope of Z (e.g., Z starts with X). The composition can then contain the operation $\textsc{exit}_X; \textsc{incr}_X$ that corresponds to $\textsc{exit1}_X$ because $\textsc{incr}_X$ is trivially enabled when the counter value of X is 0. The only other possible composition of individual operations that can appear in this case is $\textsc{exit}_X; \textsc{exit}_X$. If $\boldsymbol{min}_X = 0$, $\textsc{exit}_X; \textsc{exit}_X = \textsc{exit}_X$, which is well-defined because $\textsc{exit}_X$ is always enabled for $\boldsymbol{min}_X = 0$. If $\boldsymbol{min}_X > 0$, then $\textsc{exit}_X; \textsc{exit}_X$ is undefined, and $\textsc{exit}_X; \textsc{exit}_X$ does not contribute anything to the composition. However, this is correct since X is not nullable, and the second $\textsc{exit}_X$ is not enabled after the counter value of X is reset to 0. Intuitively, the second occurrence of X cannot be exited without iterating X at least once.

Example 4.3.3. Consider the regex R = `.*a{1,3}a{1,3}a`. Let X be the counting loop `a{1,3}`. R has two minterms `a` and `[^a]`. We get the following conditional

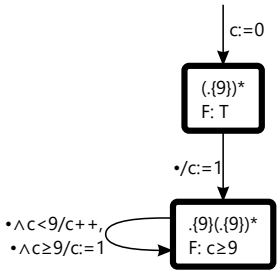


Figure 4.2: A counting automaton for the regex `(.{9})*`.

derivatives of R, starting with the case for $\partial_\alpha(S{*}Z)$ due to the normal form assumption:

$$\begin{aligned}
\partial_{\mathtt{a}}(R) &= \partial_{\mathtt{a}}(\mathtt{.}) \otimes \{\langle \textsc{id}, R\rangle\} \cup \partial_{\mathtt{a}}(XX\mathtt{a}) \\
&= \{\langle \textsc{id}, R\rangle, \langle \textsc{incr}_X, XX\mathtt{a}\rangle, \langle \textsc{exit}1_X, X\mathtt{a}\rangle\} \\
\partial_{\mathtt{a}}(XX\mathtt{a}) &= \partial_{\mathtt{a}}(\mathtt{a}) \otimes \{\langle \textsc{incr}_X, XX\mathtt{a}\rangle\} \cup \{\langle \textsc{exit}_X, \varepsilon\rangle\} \otimes \partial_{\mathtt{a}}(X\mathtt{a}) \\
&= \{\langle \textsc{incr}_X, XX\mathtt{a}\rangle\} \cup \{\langle \textsc{exit}_X, \varepsilon\rangle\} \otimes \\
&\quad \{\langle \textsc{incr}_X, X\mathtt{a}\rangle, \langle \textsc{exit}_X, \varepsilon\rangle\} \\
&= \{\langle \textsc{incr}_X, XX\mathtt{a}\rangle, \langle \textsc{exit}1_X, X\mathtt{a}\rangle\} \\
\partial_{\mathtt{a}}(X\mathtt{a}) &= \partial_{\mathtt{a}}(\mathtt{a}) \otimes \{\langle \textsc{incr}_X, X\mathtt{a}\rangle\} \cup \{\langle \textsc{exit}_X, \varepsilon\rangle\} \otimes \partial_{\mathtt{a}}(\mathtt{a}) \\
&= \{\langle \textsc{incr}_X, X\mathtt{a}\rangle, \langle \textsc{exit}_X, \varepsilon\rangle\} \\
\partial_{\mathtt{a}}(\mathtt{a}) = \partial_{\mathtt{a}}(\mathtt{.}) = \partial_{\mathtt{[\^{}a]}}(\mathtt{.}) &= \{\langle \textsc{id}, \varepsilon\rangle\} \\
\partial_{\mathtt{[\^{}a]}}(\mathtt{a}) &= \emptyset
\end{aligned}$$

Above, the composition $\textsc{exit}_X; \textsc{exit}_X$ in $\partial_{\mathtt{a}}(XX\mathtt{a})$ is undefined and thus removed. We also get that $\partial_{\mathtt{[\^{}a]}}(R) = \{\langle \textsc{id}, R\rangle\}$ where $\partial_{\mathtt{[\^{}a]}}(\mathtt{a}) = \emptyset$ and consequently $\partial_{\mathtt{[\^{}a]}}(XX\mathtt{a}) = \emptyset$ and $\partial_{\mathtt{[\^{}a]}}(X\mathtt{a}) = \emptyset$.

If we now consider, for example, the language defined by $\partial_{\mathtt{a}}(X\mathtt{a})$ in a valid counter memory $\mathfrak{m}$, it is the union of the languages $\mathbf{L}_{\textsc{incr}_X(\mathfrak{m})}(X\mathtt{a})$ and $\mathbf{L}_{\textsc{exit}_X(\mathfrak{m})}(\varepsilon)$. These correspond to the cases of continuing to iterate the loop X (if the counter value of X is below 3) or exiting the loop (if the counter value of X is at least 1) and accepting $\{\epsilon\}$. □

Example 4.3.4. Consider the regex `(.{9})*`, whose CA is in Figure 4.2. Here, '`.`' is the only input predicate and denotes the set of all characters. We explain the use of some of the counter operations in the CA of Figure 4.2 by showing how they arise through the partial-derivative-based construction of CAs as discussed above. The initial state is the regex itself. The (only) partial derivative of the state `(.{9})*` is `.{9}(.{9})*` where the body of the counting loop is exited but also incremented once, so EXIT1 is applied to c under the guard $\textsc{canExit}_c$ (which is shown as $c \geq 9/c{:=}1$ in the figure). The state `.{9}(.{9})*` has two cases of partial derivatives both leading back to `.{9}(.{9})*`.

The first case is when $c < 9$ ($\textsc{canIncr}_c$ holds), in which case c is incremented (shown as $c < 9/c{+}{+}$ in the figure). The second case is when the counting loop is conditionally nullable and is exited under the condition $\textsc{canExit}_c$ (i.e. $c \geq 9$), the value of c is reset to 0, and then c is incremented as a result of taking the partial derivative of `(.{9})*`. Thus, EXIT1 arises as a sequential composition of exiting the loop, followed

by resetting the counter to 0, and then incrementing it. Therefore, $\textsc{canExit}_c$ must hold, while the increment condition holds trivially after a reset to 0. The initial state is unconditionally final in Figure 4.2, while the other state is final only when $\textsc{canExit}_c$ holds as marked by "F :". □

We now state the correctness theorem of conditional derivatives. For that, we define $\textsc{canExit}_R$ as the predicate shown above for a normalized regex R, assuming that X stands for a counting loop.

$$\textsc{canExit}_R \stackrel{\text{def}}{=} \begin{cases} \top_{\mathbb{C}} & \text{if } R = \varepsilon, \\ \textsc{canExit}_Z & \text{else if } R = YZ \text{ and } Y \text{ is nullable}, \\ \textsc{canExit}_X \wedge \textsc{canExit}_Z & \text{else if } R = XZ, \\ \bot_{\mathbb{C}} & \text{otherwise.} \end{cases}$$

Note that Y above may also be a counting loop. However, since it is nullable, $\boldsymbol{min}_Y$ must be 0, and then $\textsc{canExit}_Y$ is always true. (If $\boldsymbol{min}_Y > 0$, then Y cannot be nullable as R is normalized.)

Theorem 4. *Let R be a normalized regex and let $\Sigma = \mathit{Minterms}(\Theta)$ where Θ is some finite superset of Π_R. If $\mathfrak{m}$ is valid for R, then $\mathbf{L}_{\mathfrak{m}}(R) = \bigcup_{\alpha\in\Sigma}[[\alpha]] \cdot \mathbf{L}_{\mathfrak{m}}(\partial_\alpha(R)) \cup \{\epsilon \mid \mathfrak{m} \models \textsc{canExit}_R\}$.*

Proof. By induction over $\#(R)$.

Base case $R = \varepsilon$. Holds because $\partial_\alpha(\varepsilon) = \emptyset$ and $\mathfrak{m} \models \top$ for any $\mathfrak{m}$.

Base case $R = \psi Z$ Here R is not nullable. We use the assumption that Σ is a set of minterms which implies that $[[\psi]] = [[\bigvee \Gamma]]$ for some $\Gamma \subseteq \Sigma$ and $\psi \wedge \alpha$ is unsatisfiable for all $\alpha \in \Sigma \setminus \Gamma$.

$$\begin{aligned} \mathbf{L}_{\mathfrak{m}}(\psi Z) &= [[\psi]] \cdot \mathbf{L}_{\mathfrak{m}}(Z) \\ &= \bigcup_{\alpha\in\Sigma} [[\alpha]] \cdot \begin{cases} \mathbf{L}_{\mathfrak{m}}(\{\langle \textsc{id}, Z\rangle\}) & \text{if } \alpha \wedge \psi \text{ is satisfiable} \\ \emptyset & \text{otherwise} \end{cases} \\ &= \bigcup_{\alpha\in\Sigma} [[\alpha]] \cdot \mathbf{L}_{\mathfrak{m}}(\partial_\alpha(\psi Z)) \\ &\overset{\textsc{canExit}_R=\bot}{=} \Big(\bigcup_{\alpha\in\Sigma} [[\alpha]] \cdot \mathbf{L}_{\mathfrak{m}}(\partial_\alpha(\psi Z))\Big) \cup \{\epsilon \mid \mathfrak{m} \models \textsc{canExit}_R\} \end{aligned}$$

Induction case $R = XZ$ where $X = S\{\ell, k\}$ is a counting loop Let $\mathfrak{m}_1 = \textsc{incr}_X(\mathfrak{m})$ and let $\mathfrak{m}_0 = \textsc{exit}_X(\mathfrak{m})$. Observe that $\mathfrak{m}_1 = \bot$ iff $\mathfrak{m}(X) = k$ and $\mathfrak{m}_0 = \bot$ iff $\mathfrak{m}(X) < \ell$. Note also that if $\mathfrak{m}_1 = \bot$ then $\mathfrak{m}_0 \neq \bot$ because $\ell \leq k$. So $\mathfrak{m}_0$ is valid for Z because $\mathfrak{m}$ is valid for XZ. Also, since $\mathfrak{m}$ is valid for R, if S contains a counter c then c is not visible R and thus $\mathfrak{m}(c) = 0$. Thus, $\mathfrak{m}$ is also valid for S.

Assume first that S is not nullable. It follows that, since $\mathfrak{m}(c) = 0$ for all $c \in \mathit{Counters}(S)$ because $\mathfrak{m}$ is valid for R and, unless $\textsc{canExit}_S = \bot$, there must be at least one counter c such that $\boldsymbol{min}_c > 0$ and $\textsc{canExit}_S$ contains the conjunct $\textsc{canExit}_c$ and so

$$\mathfrak{m} \not\models \textsc{canExit}_S \tag{4.9}$$

It is also true that

$$\begin{aligned} \mathfrak{m}_0 \neq \bot \text{ and } \mathfrak{m}_0 \models \textsc{canExit}_Z &\iff \mathfrak{m} \models \textsc{canExit}_X \text{ and } \textsc{exit}_X(\mathfrak{m}) \models \textsc{canExit}_Z \\ &\iff \mathfrak{m} \models \textsc{canExit}_{XZ} \end{aligned} \tag{4.10}$$

because $\mathit{Visible}(XZ) = \{X\} \cup (\mathit{Visible}(Z) \setminus \mathit{Counters}(S))$, so only the counters in $\{X\} \cup \mathit{Counters}(S)$ could interfere (if they occur in the scope of Z) but their value is 0 in $\mathfrak{m}_0$ by validity of $\mathfrak{m}$. Let

$$E = \{\epsilon \mid \mathfrak{m} \models \textsc{canExit}_{XZ}\}.$$

We get the following (observe that if $\mathfrak{m}_0 = \bot$, then $\mathbf{L}_{\mathfrak{m}_0}(Z) = \emptyset$):

$$\begin{aligned} \mathbf{L}_{\mathfrak{m}}(XZ) &= \mathbf{L}_{\mathfrak{m}}(S) \cdot \mathbf{L}_{\mathfrak{m}_1}(XZ) \cup \mathbf{L}_{\mathfrak{m}_0}(Z) \\ &\overset{(2\times\text{IH})}{=} \Big(\bigcup_\alpha [[\alpha]] \cdot \mathbf{L}_{\mathfrak{m}}(\partial_\alpha(S)) \cup \{\epsilon \mid \mathfrak{m} \models \textsc{canExit}_S\}\Big) \cdot \mathbf{L}_{\mathfrak{m}_1}(XZ) \cup \\ &\quad \bigcup_\alpha [[\alpha]] \cdot \mathbf{L}_{\mathfrak{m}_0}(\partial_\alpha(Z)) \cup \{\epsilon \mid \mathfrak{m}_0 \neq \bot \text{ and } \mathfrak{m}_0 \models \textsc{canExit}_Z\} \\ &\overset{((4.9),(4.10))}{=} \Big(\bigcup_\alpha [[\alpha]] \cdot \mathbf{L}_{\mathfrak{m}}(\partial_\alpha(S))\Big) \cdot \mathbf{L}_{\mathfrak{m}_1}(XZ) \cup \bigcup_\alpha [[\alpha]] \cdot \mathbf{L}_{\mathfrak{m}_0}(\partial_\alpha(Z)) \\ &\quad \cup E \\ &= \bigcup_\alpha [[\alpha]] \cdot \big(\mathbf{L}_{\mathfrak{m}}(\partial_\alpha(S)) \cdot \mathbf{L}_{\mathfrak{m}_1}(XZ) \cup \mathbf{L}_{\mathfrak{m}_0}(\partial_\alpha(Z))\big) \cup E \\ &= \bigcup_\alpha [[\alpha]] \cdot \big(\underbrace{\mathbf{L}_{\mathfrak{m}}(\partial_\alpha(S)) \cdot \mathbf{L}_{\mathfrak{m}}(\langle \textsc{incr}_X, XZ\rangle)}_{(\star) = \mathbf{L}_{\mathfrak{m}}(\partial_\alpha(S) \otimes \{\langle \textsc{incr}_X, XZ\rangle\})} \cup \mathbf{L}_{\mathfrak{m}}(\{\langle \textsc{exit}_X, \varepsilon\rangle\} \otimes \\ &\quad \partial_\alpha(Z))\big) \cup E \\ &= \bigcup_\alpha [[\alpha]] \cdot \mathbf{L}_{\mathfrak{m}}(\partial_\alpha(S) \otimes \{\langle \textsc{incr}_X, XZ\rangle\} \cup \{\langle \textsc{exit}_X, \varepsilon\rangle\} \\ &\quad \otimes \partial_\alpha(Z)) \cup E \\ &= \bigcup_\alpha [[\alpha]] \cdot \mathbf{L}_{\mathfrak{m}}(\partial_\alpha(XZ)) \cup E \end{aligned}$$

We show next that $(\star)$ holds. Let $\langle f, W\rangle \in \partial_\alpha(S)$. It suffices to show that

$$\mathbf{L}_{\mathfrak{m}}(\langle f, W\rangle) \cdot \mathbf{L}_{\mathfrak{m}}(\langle \textsc{incr}_X, XZ\rangle) = \mathbf{L}_{\textsc{incr}_X(f(\mathfrak{m}))}(WXZ)$$

holds which is the same as:

$$\mathbf{L}_{\textsc{incr}_X(f(\mathfrak{m}))}(WXZ) = \mathbf{L}_{f(\mathfrak{m})}(W) \cdot \mathbf{L}_{\textsc{incr}_X(\mathfrak{m})}(XZ) \tag{4.11}$$

Since $\mathfrak{m}$ is valid for WXZ, $\textsc{incr}_X(f(\mathfrak{m}))$ is also valid for WXZ because the potential updates to $\mathfrak{m}$ only affect visible counters. It follows that

$$\begin{aligned} \mathbf{L}_{\textsc{incr}_X(f(\mathfrak{m}))}(WXZ) &\overset{\text{(Lemma 4.1)}}{=} \mathbf{L}_{\textsc{incr}_X(f(\mathfrak{m}))}(W) \cdot \\ &\quad \cdot \mathbf{L}_{\rho(\mathit{Visible}(W))(\textsc{incr}_X(f(\mathfrak{m})))}(XZ) \\ &\overset{(X \notin \mathit{Counters}(W))}{=} \mathbf{L}_{f(\mathfrak{m})}(W) \cdot \\ &\quad \cdot \mathbf{L}_{\rho(\mathit{Visible}(W))(\textsc{incr}_X(f(\mathfrak{m})))}(XZ) \end{aligned}$$

We have that $\mathit{Visible}(W) \subseteq \mathit{Counters}(W) \subseteq \mathit{Counters}(S)$ by construction of derivatives and f only affects values of counters in $\mathit{Visible}(W)$. Since $\mathfrak{m}$ is valid for R and $\mathit{Counters}(S) \subseteq \mathit{Hidden}(R)$ it follows that $\mathfrak{m}(c) = 0$ for all $c \in \mathit{Visible}(W)$, and $X \in \mathit{Visible}(R) \setminus \mathit{Counters}(S)$. Therefore $\rho(\mathit{Visible}(W))(\textsc{incr}_X(f(\mathfrak{m}))) = \textsc{incr}_X(\mathfrak{m})$, i.e., the reset cancels the effect of f, and so

$$\mathbf{L}_{\rho(\mathit{Visible}(W))(\textsc{incr}_X(f(\mathfrak{m})))}(XZ) = \mathbf{L}_{\textsc{incr}_X(\mathfrak{m})}(XZ)$$

This completes the proof of Theorem 4.11 and ($\star$), and thus the induction case under the condition that S is not nullable. Under the condition that S is nullable, it follows that $\ell = 0$ because R is normalized. But we can pretend that S *is not nullable* because $\ell = 0$ makes X nullable and the proof remains unchanged.

Induction case $R = S{*}Z$ Observe that $\mathfrak{m} \models \textsc{canExit}_R$ iff $\mathfrak{m} \models \textsc{canExit}_Z$ because $S{*}$ is nullable and $\mathfrak{m}(c) = 0$ for all $c \in \mathit{Counters}(S)$. Assume without loss of generality that S is not nullable. In this case $\mathfrak{m} \not\models \textsc{canExit}_S$. Let $E = \{\epsilon \mid \mathfrak{m} \models \textsc{canExit}_R\}$.

$$\begin{aligned}
\mathbf{L}_{\mathfrak{m}}(S{*}Z) &= \mathbf{L}_{\mathfrak{m}}(S)^* \cdot \mathbf{L}_{\mathfrak{m}}(Z) \\
&= \mathbf{L}_{\mathfrak{m}}(S) \cdot \mathbf{L}_{\mathfrak{m}}(S{*}Z) \cup \mathbf{L}_{\mathfrak{m}}(Z) \\
&\overset{(2\times\mathrm{IH})}{=} \Big(\bigcup_{\alpha} [\![\alpha]\!] \cdot \mathbf{L}_{\mathfrak{m}}(\partial_\alpha(S)) \cup \{\epsilon \mid \mathfrak{m} \models \textsc{canExit}_S\}\Big) \cdot \mathbf{L}_{\mathfrak{m}}(S{*}Z) \cup \\
&\quad \bigcup_{\alpha} [\![\alpha]\!] \cdot \mathbf{L}_{\mathfrak{m}}(\partial_\alpha(Z)) \cup \{\epsilon \mid \mathfrak{m} \models \textsc{canExit}_Z\} \\
&= \bigcup_{\alpha} [\![\alpha]\!] \cdot \mathbf{L}_{\mathfrak{m}}(\partial_\alpha(S)) \cdot \mathbf{L}_{\mathfrak{m}}(S{*}Z) \cup \bigcup_{\alpha} [\![\alpha]\!] \cdot \mathbf{L}_{\mathfrak{m}}(\partial_\alpha(Z)) \cup E \\
&= \bigcup_{\alpha} [\![\alpha]\!] \cdot \big(\mathbf{L}_{\mathfrak{m}}(\partial_\alpha(S)) \cdot \mathbf{L}_{\mathfrak{m}}(S{*}Z) \cup \mathbf{L}_{\mathfrak{m}}(\partial_\alpha(Z))\big) \cup E \\
&= \bigcup_{\alpha} [\![\alpha]\!] \cdot \Big(\bigcup_{\langle f, W\rangle \in \partial_\alpha(S)} \mathbf{L}_{f(\mathfrak{m})}(W) \cdot \mathbf{L}_{\mathfrak{m}}(S{*}Z) \cup \mathbf{L}_{\mathfrak{m}}(\partial_\alpha(Z))\Big) \cup E \\
&\overset{(\star\star)}{=} \bigcup_{\alpha} [\![\alpha]\!] \cdot \Big(\bigcup_{\langle f, W\rangle \in \partial_\alpha(S)} \mathbf{L}_{f(\mathfrak{m})}(WS{*}Z) \cup \mathbf{L}_{\mathfrak{m}}(\partial_\alpha(Z))\Big) \cup E \\
&= \bigcup_{\alpha} [\![\alpha]\!] \cdot \big(\mathbf{L}_{\mathfrak{m}}(\partial_\alpha(S) \otimes \{\langle \textsc{id}, S{*}Z\rangle\}) \cup \mathbf{L}_{\mathfrak{m}}(\partial_\alpha(Z))\big) \cup E \\
&= \bigcup_{\alpha} [\![\alpha]\!] \cdot \mathbf{L}_{\mathfrak{m}}(\partial_\alpha(S) \otimes \{\langle \textsc{id}, S{*}Z\rangle\} \cup \partial_\alpha(Z)) \cup E \\
&= \bigcup_{\alpha} [\![\alpha]\!] \cdot \mathbf{L}_{\mathfrak{m}}(\partial_\alpha(S{*}Z)) \cup E
\end{aligned}$$

Equality ($\star\star$) holds by using Lemma 4.1 because $\rho(\mathit{Visible}(W))(f(\mathfrak{m})) = \mathfrak{m}$ since $\mathfrak{m}(c) = 0$ for $c \in \mathit{Counters}(S)$ and $\mathit{Visible}(W) \subseteq \mathit{Counters}(W) \subseteq \mathit{Counters}(S)$ by definition of conditional derivatives.

Induction case $R = (Y_1|Y_2)Z$ In this case $\mathfrak{m} \models \textsc{canExit}_{Y_i Z}$ iff Y_i is nullable and $\mathfrak{m} \models \textsc{canExit}_Z$ because $\mathfrak{m}(c) = 0$ for $c \in \mathit{Counters}(Y_i)$.

$$
\begin{aligned}
\mathbf{L}_{\mathfrak{m}}((Y_1|Y_2)Z) &= \mathbf{L}_{\mathfrak{m}}(Y_1Z) \cup \mathbf{L}_{\mathfrak{m}}(Y_2Z) \\
&\overset{(2\times\text{IH})}{=} \bigcup_{i=1,2} \Big(\bigcup_{\alpha} [[\alpha]] \cdot \mathbf{L}_{\mathfrak{m}}(\partial_\alpha(Y_iZ)) \cup \{\epsilon \mid \mathfrak{m} \models \textsc{canExit}_{Y_iZ}\}\Big) \\
&= \bigcup_{\alpha} [[\alpha]] \cdot \mathbf{L}_{\mathfrak{m}}(\partial_\alpha(Y_1Z) \cup \partial_\alpha(Y_2Z)) \cup \{\epsilon \mid \mathfrak{m} \models \textsc{canExit}_{Y_1Z} \vee \\
&\qquad \textsc{canExit}_{Y_2Z}\}\Big) \\
&= \bigcup_{\alpha} [[\alpha]] \cdot \mathbf{L}_{\mathfrak{m}}(\partial_\alpha((Y_1|Y_2)Z)) \cup \{\epsilon \mid \mathfrak{m} \models \textsc{canExit}_{R}\}\Big)
\end{aligned}
$$

The last equality uses that $\mathfrak{m} \models \textsc{canExit}_{Y_1Z} \vee \textsc{canExit}_{Y_2Z}$ iff $\mathfrak{m} \models \textsc{canExit}_{(Y_1|Y_2)Z}$. The statement follows by the induction principle. □

4.3.3 Constructing CAs from Conditional Derivatives

We convert a normalized regex R to the counting automaton $\mathrm{CA}(R)$ whose set of states is the smallest set containing R as the initial state and all those regexes that arise in conditional derivatives constructed from R by repeated derivation wrt Σ. Given a state represented by a regex S, for each $\alpha \in \Sigma$ and each partial conditional derivative $\langle f, T\rangle \in \partial_\alpha(S)$, there is a transition $S \text{-}\{\alpha, f\}\!\!\rightarrow T$ in $\mathrm{CA}(R)$. The *final condition* $F(S)$ of a state S of $\mathrm{CA}(R)$ is $\textsc{canExit}_S$. Observe that $F(S) = \bot_{\mathbb{C}}$ when S is not nullable and has no visible counters, which corresponds to the classical case.

Theorem 5. *Let R be a normalized regex and $A = FA(CA(R))$. Then, for all $\langle \mathfrak{m}, S\rangle \in Q_A$, $\mathcal{L}_A(\langle \mathfrak{m}, S\rangle) = \mathbf{L}_{\mathfrak{m}}(S)$.*

Proof. The following fundamental equation is a characterization of the language of a state q:

$$
\mathcal{L}_A(q) = \Big(\bigcup_{(q,\alpha,p)\in\Delta_A} [[\alpha]] \cdot \mathcal{L}_A(p)\Big) \cup \{\epsilon \mid q \in F\} \tag{4.12}
$$

(i.e., $\epsilon \in \mathcal{L}_A(q)$ iff $q \in F$). We write $\mathcal{L}(q)$ for $\mathcal{L}_A(q)$ when A is clear from the context.

Let R be fixed. We prove the following statement by induction over the length of w:

$$
\forall \langle \mathfrak{m}, S\rangle \in Q_A : w \in \mathcal{L}_A(\langle \mathfrak{m}, S\rangle) \iff w \in \mathbf{L}_{\mathfrak{m}}(S)
$$

Base case $w = \epsilon$ Fix $q = \langle \mathfrak{m}, S\rangle \in Q_A$. Then $\epsilon \in \mathcal{L}_A(q)$ iff (by Equation 4.12) $q \in F_A$ iff (by definition of $F_{\mathrm{CA}(R)}$) $\mathfrak{m} \models \textsc{canExit}_S$ iff (by Theorem 4) $\epsilon \in \mathbf{L}_{\mathfrak{m}}(S)$.

Induction case $w = av$ Fix $\langle \mathfrak{m}, S\rangle \in Q_A$. Choose the unique $\alpha \in \Sigma$ such that $a \in [[\alpha]]$.

$$
\begin{aligned}
av \in \mathbf{L}_{\mathfrak{m}}(S) &\overset{\text{(Theorem 4)}}{\iff} av \in [[\alpha]] \cdot \mathbf{L}_{\mathfrak{m}}(\partial_\alpha(S)) \\
&\iff \exists \langle f, T\rangle \in \partial_\alpha(S) : v \in \mathbf{L}_{f(\mathfrak{m})}(T) \\
&\overset{\text{(IH)}}{\iff} \exists \langle f, T\rangle \in \partial_\alpha(S) : v \in \mathcal{L}_A(\langle f(\mathfrak{m}), T\rangle) \\
&\iff \exists \langle \mathfrak{m}, S\rangle \text{-}\{\alpha\}\!\!\rightarrow \langle f(\mathfrak{m}), T\rangle \in \Delta_A : v \in \mathcal{L}_A(\langle f(\mathfrak{m}), T\rangle) \\
&\overset{(4.12)}{\iff} av \in \mathcal{L}_A(\langle \mathfrak{m}, S\rangle)
\end{aligned}
$$

the statement follows by the induction principle. □

The construction of $\mathrm{CA}(R)$ terminates, and the number of states of $\mathrm{CA}(R)$ is linear in $\#_\Psi(R)$.

Theorem 6. *Let R be a normalized regex. Then $|Q_{CA(R)}| \leq \#_\Psi(R) + 1$.*

To prove Theorem 6 we first introduce the following notions. Let $\partial^+(R)$ denote the set of all regexes arising through partial derivatives applied recursively starting from a normalized regex R. Formally, let:

$$\partial(R) \stackrel{\text{def}}{=} \{W \mid \exists \alpha, f : \alpha \in \mathit{Minterms}(R), \langle f, W\rangle \in \partial_\alpha(R)\},$$

then $\partial^+(R)$ is the least fixed point of the following equations, where L is a set of normalized regexes:

$$\partial^+(R) = \partial(R) \cup \partial^+(\partial(R)), \quad \partial^+(L) = \bigcup_{R \in L} \partial^+(R).$$

We first prove the following lemma where given a set of normalized regexes L and a normalized regex Z we let LZ denote the set $\{WZ \mid W \in L\}$ of normalized regexes. Observe that $\partial^+(R)$ is the set of regexes reached after one or more derivations, which may but need not include R itself, e.g., $\partial^+(\texttt{b(ab)\{9\}}) = \{\texttt{(ab)\{9\}}, \texttt{b(ab)\{9\}}\}$ includes the start regex while $\partial^+(\texttt{ab}) = \{\texttt{b}, \varepsilon\}$ does not. We write $S\diamond$ for a counting loop $S\{\ell, k\}$ or loop $S*$.

Lemma 4.3. *Let X and Z be any normalized regexes. Then $\partial^+(XZ) = \partial^+(X)Z \cup \partial^+(Z)$ and if X is a (counting) loop $S\diamond$ then $\partial^+(X) = \partial^+(S)X$.*

Proof. We prove by induction over $\#(X)$ that $\partial^+(XZ) = \partial^+(X)Z \cup \partial^+(Z)$. The base case $X = \varepsilon$ follows immediately because $\partial(\varepsilon) = \emptyset$.

Induction case $X = \psi Y$:

$$\begin{aligned}
\partial^+(XZ) &= \{YZ\} \cup \partial^+(YZ) \\
&\stackrel{IH}{=} \{YZ\} \cup \partial^+(Y)Z \cup \partial^+(Z) \\
&= (\{Y\} \cup \partial^+(Y))Z \cup \partial^+(Z) \\
&= \partial^+(X)Z \cup \partial^+(Z)
\end{aligned}$$

Induction case $X = (X_1|X_2)Y$:

$$\begin{aligned}
\partial^+(XZ) &= \partial^+(X_1YZ) \cup \partial^+(X_2YZ) \\
&\stackrel{2\times IH}{=} \partial^+(X_1Y)Z \cup \partial^+(X_2Y)Z \cup \partial^+(Z) \\
&\stackrel{2\times IH}{=} (\partial^+(X_1)Y \cup \partial^+(Y))Z \cup (\partial^+(X_2)Y \cup \partial^+(Y))Z \cup \partial^+(Z) \\
&= (\partial^+(X_1)Y \cup \partial^+(X_2)Y \cup \partial^+(Y))Z \cup \partial^+(Z) \\
&= (\partial^+(X_1|X_2)Y \cup \partial^+(Y))Z \cup \partial^+(Z) \\
&\stackrel{(\star)}{=} \partial^+(X)Z \cup \partial^+(Z)
\end{aligned}$$

In $(\star)$, if $Y = \varepsilon$, the equality holds by definition of derivatives of a choice node. If $Y \neq \varepsilon$, then $X_1|X_2$ is smaller than X, and one can apply the IH on $(X_1|X_2)Y$ with $X_1|X_2$ as X and Y as an instance of the universal variable Z in the lemma.

Induction case of $X = S\diamond Y$ where $S\diamond$ is either a counting loop or a $*$-loop. the proof step uses the property that, for any normalized W:

$$\partial^+(S\diamond W) = \partial^+(S)S\diamond W \cup \partial^+(W) \tag{4.13}$$

holds. Equation 4.13 is proved as follows:

$$\begin{aligned}
\partial^+(S\diamond W) &= \partial^+(SS\diamond W) \cup \partial^+(W) \\
&\overset{\text{(IH)}}{=} \partial^+(S)S\diamond W \cup \partial^+(S\diamond W) \cup \partial^+(W) \\
&\overset{\text{(lfp)}}{=} \partial^+(S)S\diamond W \cup \partial^+(W)
\end{aligned}$$

where (lfp) holds because $\partial^+(S\diamond W) \subseteq \partial^+(S)S\diamond W \cup \partial^+(W)$ that can be shown separately. It follows that:

$$\begin{aligned}
\partial^+(XZ) &= \partial^+(S\diamond(YZ)) \\
&\overset{(4.13)}{=} \partial^+(S)S\diamond YZ \cup \partial^+(YZ) \\
&\overset{IH}{=} \partial^+(S)S\diamond YZ \cup \partial^+(Y)Z \cup \partial^+(Z) \\
&= (\partial^+(S)S\diamond Y \cup \partial^+(Y))Z \cup \partial^+(Z) \\
&\overset{(4.13)}{=} \partial^+(S\diamond Y)Z \cup \partial^+(Z) \\
&= \partial^+(X)Z \cup \partial^+(Z)
\end{aligned}$$

The statement follows by the induction principle. Observe that Lemma 4.13 implies the second part of the lemma by letting $W = \varepsilon$. □

Now are ready to prove Theorem 6. Recall that $|S|$ is the cardinality of a set S, and $\#(R)$ is the size of a regex R that is the number of abstract syntax tree nodes of R (up to the case of $R = \epsilon$ where the size is 0), and $\#_\Psi(R)$ is the number of predicates nodes in R.

Proof of Theorem 6. We first prove, by induction over $\#(R)$, that Theorem 4.14 holds.

$$|\partial^+(R)| \leq \#_\Psi(R) \tag{4.14}$$

Base case $R = \varepsilon$ Then $\partial^+(R) = \emptyset$ and $\#_\Psi(R) = 0$. Theorem 4.14 holds trivially.
Induction case $R = \psi Z$ This gives us $\partial^+(R) = \{Z\} \cup \partial^+(Z)$. Thus

$$|\partial^+(R)| \leq |\partial^+(Z)| + 1 \overset{\text{IH}}{\leq} \#_\Psi(Z) + 1 = \#_\Psi(R).$$

Induction case $R = (X|Y)Z$ This gives us $\partial^+(R) = \partial^+(XZ) \cup \partial^+(YZ)$. Then, by Lemma 4.3,

$$\partial^+(XZ) \cup \partial^+(YZ) = \partial^+(X)Z \cup \partial^+(Y)Z \cup \partial^+(Z)$$

which implies that (observe that, trivially, $|LZ| = |L|$ for any set L and regex Z)

$$|\partial^+(R)| \leq |\partial^+(X)| + |\partial^+(Y)| + |\partial^+(Z)| \overset{\text{IH}}{\leq} \#_\Psi(X) + \#_\Psi(Y) + \#_\Psi(Z) = \#_\Psi(R).$$

Induction case $R = S \diamond Z$ Here $\partial^+(R) = \partial^+(S)S \diamond Z \cup \partial^+(Z)$ by using Lemma 4.13 in Lemma 4.3. Thus,

$$|\partial^+(R)| \leq |\partial^+(S)| + |\partial^+(Z)| \leq^{IH} \#_\Psi(S) + \#_\Psi(Z) = \#_\Psi(R).$$

Equation 4.14 follows by the induction principle. Theorem 6 follows from Theorem 4.14 because $Q_{\mathrm{CA}(R)} = \partial^+(R) \cup \{R\}$ and thus $|Q_{\mathrm{CA}(R)}| \leq |\partial^+(R)| + 1 \leq \#_\Psi(R) + 1$. □

We get the following final correctness result as a corollary of Theorem 5, Theorem 3, and Theorem 6.

Corollary 1. *Let R be a normalized regex. Then* $L(R) = L(CA(R))$.

Proof. First, $Q_{\mathrm{CA}(R)}$ is finite, and thus well-defined by using Theorem 6. Use Theorem 5 with $\langle \mathfrak{m}, S \rangle$ as the initial state $\langle \mathbf{0}, R \rangle$ of A. It follows that $L(A) = \mathbf{L_0}(R)$. Then use Theorem 3 for $\mathbf{L_0}(R) = L(R)$ and $L(\mathrm{CA}(R)) = L(A)$ holds by definition. □

A further important aspect of $\mathrm{CA}(R)$ is that, although the number of input minterms may potentially be exponential in the number of predicates in R, in the case of predicates being represented as a finite union of intervals (as is done typically for character classes), the size of a single predicate representation can be estimated to be proportional to the number of interval borders in the union. In this case, the total size of all the minterms remains linear in the total size of all the predicates because the total number of interval borders will remain the same in minterms as in the original set of predicates. In other words, mintermization based on character classes does not blow up the number of transition in $\mathrm{CA}(R)$. We have also validated this fact experimentally.

4.4 From Counting Automata to Counting-Set Automata

CAs obtained through conditional derivatives as shown in Section 4.3.3 are nondeterministic. As the major contributions of this thesis, we now propose an approach for determinising them into a form that can be used efficiently for regex matching.

The approach from which we start and to which we contrast our new method is the naive determinisation of CAs to DFAs. The given CA is first converted to its underlying NFA, by making the counter memories an explicit part of control states. The NFA is in turn determinised by the textbook subset construction.

Already the first step, the construction of the NFA, oftentimes explodes since it sacrifices the succinctness of symbolic counters (it is linear to the counter bounds). This initial blow-up is then much amplified in the subset construction, which is exponential to the size of the NFA and hence also to the repetition bounds (as, e.g., in the case of the regex `.*a.{`k`}` with its CA in Figure 4.1a).

Our answer to this problem is a direct determinisation of the CA into a novel type of automata, which we call *counting-set automata (CsAs)*. Control states of counting-set automata produced by our determinisation are essentially the states of the corresponding DFA but with the counter memories removed. In order to be able to simulate a run of the DFA, they are equipped with special registers that can hold *sets* of integers, and they use them to compute the right counter memories at runtime. This completely avoids the state space explosion of the naive construction caused by wiring counter memories into control states. Moreover, the simulation is fast because all the manipulations with a counting set can be done in constant time.

4.4.1 Counting-Set Automata

We now formalize the idea of counting-set automata outlined above. We use the notion of a combined Boolean algebra $\mathbb{I}\times\mathbb{S}$, which allows us to manipulate pairs of predicates from the input algebra $\mathbb{I}$ and the counting-set algebra $\mathbb{S}$. For the purposes of this thesis, we assume that predicates in $\Psi_{\mathbb{I}\times\mathbb{S}}$ have the form $\alpha \wedge \beta$ where $\alpha \in \Psi_{\mathbb{I}}$ and $\beta \in \Psi_{\mathbb{S}}$. The conjuction $(\alpha \wedge \beta) \wedge_{\mathbb{I}\times\mathbb{S}} (\alpha' \wedge \beta')$ has the usual meaning of $(\alpha \wedge_{\mathbb{I}} \alpha') \wedge (\beta \wedge_{\mathbb{S}} \beta')$ and $\alpha \wedge \beta$ is satisfiable if both α and β are satisfiable in their respective algebras.

Counting sets. We consider a set-based interpretation of counters where the value of a counter c is a *finite set* rather than a single value. A counter under such an interpretation is referred to as a *counting set*. A *(counting-)set memory* for C is a function $\mathfrak{s} : C \to \mathcal{P}_{\text{fin}}(\mathbb{N})$ such that, for all $c \in C$, $Max(\mathfrak{s}(c)) \leq \boldsymbol{max}_c$.[2] Observe that the set of all set memories for C is *finite*. Counting-set predicates over C form an effective Boolean algebra $\mathbb{S}_C$ called the *counting-set algebra over* C, also denoted just $\mathbb{S}$ when C is clear from the context, whose domain $\Sigma_{\mathbb{S}}$ is the set of all set memories for C. The set of predicates $\Psi_{\mathbb{S}}$ is the Boolean closure of the basic predicates canIncr_c and canExit_c, hence syntactically the same as in the counter algebra $\mathbb{C}$, but with a different semantics under $\mathbb{S}$:

$$\mathfrak{s} \models \text{canExit}_c \iff Max(\mathfrak{s}(c)) \geq \boldsymbol{min}_c \text{ and } \mathfrak{s} \models \text{canIncr}_c \iff Min(\mathfrak{s}(c)) < \boldsymbol{max}_c$$

where $Min(\cdot)$ and $Max(\cdot)$ are the set minimum and maximum, respectively. Intuitively, the conditions test existence of a set element satisfying the same counter condition.

Counting-set automata. A *counting-set automaton* (CsA) is a tuple $A = (\mathbb{I}, C, Q, q_0, F, \Delta)$ where: $\mathbb{I}$ is an effective Boolean algebra called the *input algebra*. C is a finite set of *counters* associated with the counting-set algebra $\mathbb{S}$. Q is a finite set of *states* with $q_0 \in Q$ being the *initial state*. $F : Q \to \Psi_{\mathbb{S}}$ is the *final-state condition*. $\Delta \subseteq Q \times \Psi_{\mathbb{I}\times\mathbb{S}} \times (C \to \mathcal{P}(O)) \times Q$ is a finite set of *transitions*. The second component is its *guard*. The third component is the *counting-set operator* in which $O = \{\text{Incr}, \text{Noop}, \text{Rst}, \text{Rst1}\}$ is the set of *counting-set operations*. They are essentially counter operations lifted to sets (note the use of the larger initial letters to distinguish them from the counter operations). We also use the different names Rst and Rst1 for the lifting of exit and exit1 to stress their different usage (not only for exiting a loop but also for initialisation when entering the loop as will become clear in Equation 4.18).

The CsA A is *deterministic* iff the following holds for every two transitions $p \text{-}\{\psi_1, f_1\}\!\!\to q_1$ and $p \text{-}\{\psi_2, f_2\}\!\!\to q_2$ in Δ: if $\psi_1 \wedge \psi_2$ is satisfiable, then $f_1 = f_2$ and $q_1 = q_2$.

Semantics of CsAs. The semantics of an indexed counting-set operation $\text{op}_c \in O$ is the set transformer $upd(\text{op}_c)$ defined as follows:

$$upd(\text{Incr}_c) = \boldsymbol{\lambda} S.\{n+1 \mid n \in S \wedge n < \boldsymbol{max}_c\} \qquad upd(\text{Rst}_c) = \boldsymbol{\lambda} S.\{0\}$$
$$upd(\text{Noop}_c) = \boldsymbol{\lambda} S.S \qquad upd(\text{Rst1}_c) = \boldsymbol{\lambda} S.\{1\}$$

[2] We write $\mathcal{P}_{\text{fin}}(X)$ for the powerset of X restricted to finite nonempty sets.

Then, the counting-set operator $f : C \to \mathcal{P}(O)$ is assigned the counting-set-memory transformer $\mathbf{f} : \Sigma_{\mathbb{S}} \to \Sigma_{\mathbb{S}}$ defined as follows:

$$\mathbf{f}(\mathfrak{s}) \stackrel{\text{def}}{=} \lambda c. \begin{cases} \bigcup_{\text{OP} \in f(c)} upd(\text{OP}_c)(\mathfrak{s}(c)) & \text{if } f(c) \neq \emptyset \\ \{0\} & \text{if } f(c) = \emptyset \end{cases} \tag{4.15}$$

That is, (1) if $f(c) \neq \emptyset$, then the value $\mathfrak{s}(c)$ of each counting set c is transformed into the union of the counting sets that result from applying the operations from $f(c)$ on $\mathfrak{s}(c)$, and (2) if $f(c) = \emptyset$, then c is implicitly reset to $\{0\}$ (an implicit Rst). Our determinisation procedure creates such transitions when the value of c is irrelevant (when c is a dead variable). Let $\tau \in \Delta$. We use $\mathbf{f}_\tau(\mathfrak{s})$ to denote that the transformer is applied only on the counter-set operator of a transition τ.

Note that, unlike counter operators of a CA, a counting-set operator f does not induce any guard. The guard is rather a separate component of the transition. This is because CsA transitions produced in the CA-to-CsA determinisation need guards that are partially independent of the operations of f. In particular, we will we need to distinguish cases such as $\neg\text{canExit}_c \wedge \text{canIncr}_c$, $\text{canExit}_c \wedge \neg\text{canIncr}_c$, or $\text{canExit}_c \wedge \text{canIncr}_c$. The guard hence cannot be induced by f alone.

Note also that, unlike in CAs, the updates are defined for *indexed* operations. The reason is that the semantics of the Incr operation is restricted to never produce values greater than $\boldsymbol{max}_c$.

Finally, the *language of the CsA A* is defined through its underlying *configuration FA*, $FA(A)$, as $L(A) := L(FA(A))$. The states of $FA(A)$ are *configurations* of A, namely, tuples of the form $(q, \mathfrak{s}) \in Q \times \Sigma_{\mathbb{S}}$ consisting of a state q and a counting-set memory $\mathfrak{s}$. There are finitely many such configurations. The initial state of $FA(A)$ is the *initial configuration* $(q_0, \{c \mapsto \{0\}\}_{c \in C})$ of A. A transition $\tau = p \text{-}\{\alpha \wedge \beta, f\}\!\!\to q \in \Delta$ is *enabled* in a configuration $(p, \mathfrak{s})$ iff α is satisfiable and $\mathfrak{s} \in [\![\beta]\!]_{\mathbb{S}}$, meaning that $\mathfrak{s}$ satisfies the counter guard β. If τ is enabled in $(p, \mathfrak{s})$, then $FA(A)$ contains the transition $(p, \mathfrak{s}) \text{-}\{\alpha\}\!\!\to (q, \mathbf{f}(\mathfrak{s}))$. Finally, a state $(q, \mathfrak{s})$ of $FA(A)$ is *final* iff $\mathfrak{s} \models F(q)$.

Runtime efficiency of counting sets. A major reason for choosing CsAs as the target kind of machine for determinisation of CAs is that pattern matching with CsAs is fast. As explained in Section 4.1, the counting-set data structure can be implemented efficiently. Here, it remains to point out that the counting-set algebra and its operations can be implemented directly over that data structure. Namely, all counting-set tests and updates, and their combinations—canIncr_c, canExit_c, Noop, Incr, Rst, and Rst1—can easily be implemented through the operations of the counting-set data structure and can then run in constant time, regardless of the size of the counting set and the value $\boldsymbol{max}_c$ (assuming constant-time complexity of integer arithmetic operations).

Example 4.4.1. An example of a CsA is in Figure 4.1b. It uses intuitive notations that were also introduced in Section 4.1 as abbreviations for the operations of the counting-set data structure. Counting-set operators are depicted as assignments to c, Rst is represented as $\{0\}$ on the right of the assignment, Rst1 is represented by $\{1\}$, Incr by $c + 1$, and Noop by c. Multiple transitions between the same states and with the same updates are merged into one with a simplified guard. An example whose notation closely follows the formal development is in Figure 4.3. □

When processing a single letter of some text in pattern matching, tests and updates of one counting set then take $O(1)$ time, which in turn gives $O(|C|)$ for all counting sets. *This is our major achievement: the independence of the running time from the repetition bounds.*

4.4.2 Encoding DFA Powerstates as CsA Configurations

In order to build intuition needed for understanding our determinisation algorithm, we will first concretize how the configurations of a CsA can encode states of a DFA corresponding to the NFA $FA(A)$ underlying a given CA $A = (\mathbb{I}, C, Q, q_0, F, \Delta)$. First, recall that, since A is converted into $FA(A)$ by making the counter memories explicit parts of control states, the states of $FA(A)$ are pairs $(p, \mathfrak{m})$ consisting of a state p of A and a counter memory $\mathfrak{m}$. Second, assume that $FA(A)$ is determinised using the textbook subset construction.[3] We denote the result as $DFA(A)$ from now on. Then, the states of $DFA(A)$ are sets of states of $FA(A)$, i.e., sets of pairs $(p, \mathfrak{m})$, which we will call *powerstates*. The control states of the CsA A' built by our CA-to-CsA determinisation will be subsets of the set Q of states of the CA A. The configurations of A' will thus be pairs $(R, \mathfrak{s})$ where $R \subseteq Q$ is a CsA control state, i.e., a set of states of A, and $\mathfrak{s} : C \to \mathcal{P}_{\text{fin}}(\mathbb{N})$ is a counting-set memory. Let us now consider how $\mathfrak{s}$ can be interpreted in this context.

Naive encoding. A naive interpretation of a CsA configuration $(R, \mathfrak{s})$ is a DFA state containing all pairs $(r, \mathfrak{m})$ such that $r \in R$ and, for all $c \in C$, $\mathfrak{m}(c)$ can be any value from $\mathfrak{s}(c)$. The set of the counter memories $\mathfrak{m}$ is then isomorphic to the Cartesian product $\prod_{c \in C} \mathfrak{s}(c)$ of the sets $\mathfrak{s}(c)$ assigned to the counters, and the entire powerstate is the Cartesian product $R \times \mathfrak{m}$ of the set of states and the set of counter memories. The naive interpretation, however, is too impractical as it cannot express any dependence of a counter memory on the CA state (every state can be paired with each considered memory) nor any mutual dependence of values of different counters within a counter memory (every possible value of a counter c can be paired with every possible value of any other counter d). Most DFAs compiled from real-life regexes do not fit into this representation. For instance, the DFA configuration $\{(q, c = 0), (s, c = 0), (s, c = 1)\}$ of the CA from Figure 4.1 in Section 4.1 could not be represented by a CsA configuration because q and s appear with different sets of counter values.

Encoding with counter scopes. Our key observation how to resolve the above problem (at least for many real-life scenarios) is to take advantage of that not every counter is "used" at every CA state. In fact, the value of a counter is usually implicitly 0 at most states except a few. If these states are known, the implicit zeros do not have to be remembered explicitly in the counting sets, and the encoding becomes much more flexible. To formalize this, we introduce the notion of the *scope of a counter* that over-approximates the set of states where a counter c can have a non-zero value and

[3] The DFA produced by the textbook subset construction from a *simple* FA $\mathcal{A} = (\mathbb{I}, Q, q_0, F, \Delta)$ will have $\mathcal{P}(Q)$ as the set of states, transitions $S \text{-}\{\alpha\}\!\!\rightarrow \{r \in Q \mid s \text{-}\{\alpha\}\!\!\rightarrow r \in \Delta, s \in S\}$, the initial state $\{q_0\}$, and as the final states all those intersecting F. We note that to determinise a CA which is not simple, one could start from the more sophisticated version of the subset construction for symbolic automata of [103], which avoids explicit generation of all minterms.

that is easy to compute.[4] The scope is defined inductively as the smallest set of states $\sigma(c)$ as follows:

1. $q \in \sigma(c)$ if there is a transition to q with either $\textsc{incr}_c$ or $\textsc{exit1}_c$, or
2. there is a transition to q from a state in $\sigma(c)$ with the $\textsc{noop}_c$ operation.

In other words, the scope of c spreads from an increment of c along the transition relation until a transition with $\textsc{exit}_c$.

The DFA *powerstate encoded by a CsA configuration* $(R, \mathfrak{s})$ can then be formally defined as the set $(R, \mathfrak{s})^{DFA}$ of configurations $(r, \mathfrak{m})$ of the CA A such that $r \in R$ and, for all $c \in C$, $\mathfrak{m}(c) \in \mathfrak{s}(c)$ if $c \in \sigma(r)$, else $\mathfrak{m}(c) = 0$. We call the powerstates of $DFA(A)$ that can be encoded by CsA configurations *Cartesian*, and call the entire DFA Cartesian if all its powerstates are Cartesian.

Example 4.4.2. The powerstates of the $DFA(A)$ of the CA A from Figure 4.1a are indeed Cartesian (as discussed in Section 4.1) because q_0 is not in the scope of c. The encoding of powerstates by CsA configurations is also illustrated in Section 4.1 and later also in Example 4.4.3. □

The Cartesian encoding still cannot express all kinds of DFA powerstates. In particular, it cannot express more subtle dependencies of counter values on the state, and dependencies of counter values of different counters on each other, which mainly concerns CAs with nested counting loops compiled from regexes with nested counting sub-expressions. Example 4.4.4 discusses a regex that leads to a non-Cartesian CA. However, we later present a strong empirical evidence that a significant majority of real-life regexes lead to Cartesian CA.

4.4.3 Generalized Subset Construction

We will now describe the core of our CA-to-CsA determinisation. It is built on top of the textbook subset construction for NFAs. We use the CA from Figure 4.3a as a running example through the section. We make a simplifying assumption that the input CAs are simple (different character classes on their transitions do not overlap). This is satisfied by CAs generated by the derivative construction from Section 4.3 since their transitions are labeled by minterms of the original regex. The assumption could be dropped and the construction could be relatively easily generalized in the style of symbolic automata determinisation of [103].

Let $A = (\mathbb{I}, C, Q, q_0, F, \Delta)$ be a simple CA with the scope function $\sigma : Q \to \mathcal{P}(C)$. The algorithm produces the deterministic CsA $A' = (\mathbb{I}, C, Q', S_0, F', \Delta')$ whose components are constructed as described below. Namely, control states of A', called powerstates, are subsets of Q, i.e., $Q' \subseteq \mathcal{P}(Q)$. The initial powerstate is $S_0 = \{q_0\}$. A powerstate $S \in Q'$ is final iff the final condition holds for some of its elements, i.e.,

[4] Computing the precise set of states where a counter c can have a non-zero value would require a reachability analysis in the general case (since some of the transitions may never be executable—think of simultaneously counting with counters c and d such that $\textsc{canIncr}_c < \textsc{canExit}_d$, then the exit transition for d will never be taken). For the CAs coming from our derivative construction, the scope, however, corresponds to this set precisely—indeed, no transitions that would never be executable are generated.

$F'(S) \stackrel{\text{def}}{=} \bigvee_{q \in S} F(q)$. The sets Δ' and Q' are constructed by a fixpoint computation that explores the state space reachable from S_0. During the construction, transitions starting from previously reached powerstates are constructed and included together with their target states into Δ' and Q', respectively, until no new powerstates can be reached.

Transitions starting from a given control state R of the CsA A' are constructed to update the runtime values of counting sets such that they simulate transitions of the DFA corresponding to the CA A. Assume a CsA configuration $(R, \mathfrak{s})$ and a DFA transition $(R, \mathfrak{s})^{DFA} \text{-}\{\alpha\}\!\mapsto P$ from the DFA powerstate encoded by $(R, \mathfrak{s})$ over an input minterm α. The simulating CsA transition must transform $(R, \mathfrak{s})$ into $(R', \mathfrak{s}')$ with $(R', \mathfrak{s}')^{DFA} = P$. The simulated DFA transition was constructed from α-transitions of the NFA $FA(A)$ that are actually instantiations of the CA α-transitions enabled in configurations $(r, \mathfrak{m}) \in (R, \mathfrak{s})^{DFA}$. The simulating CsA transition will be constructed from these CA transitions. They can be identified by (1) their source state, which must be in R, (2) an alphabet minterm $\alpha \in \Sigma$ where Σ is the set of minterms over all input predicates in the CA A, and (3) their compatibility with a particular set of enabled/disabled counter guards. This set of guards belongs to the set of minterms $\Gamma_{R,\alpha}$ of the set of counter guards on the α-transitions originating in R:

$$\Gamma_{R,\alpha} \stackrel{\text{def}}{=} \mathit{Minterms}(\{grd(\text{OP}_c) \mid r \text{-}\{\alpha, f\}\!\mapsto s \in \Delta, r \in R \wedge c \in \sigma(r), \text{OP}_c \in f\}) \quad (4.16)$$

Hence, the CsA will have a transition leaving R for each $\alpha \in \Sigma$ and $\beta \in \Gamma_{R,\alpha}$, and the transition will be built from the set of CA α-transitions originating in R and *consistent* with β:

$$\Delta_{R,\alpha,\beta} \stackrel{\text{def}}{=} \{r \text{-}\{\alpha, f\}\!\mapsto s \in \Delta \mid r{\in}R, \mathbf{\mathit{Sat}}(\varphi_f \wedge \beta)\}. \quad (4.17)$$

The target of the transition is the set T of all target states of the transitions in $\Delta_{R,\alpha,\beta}$, and its guard is $\alpha \wedge \beta$.[5]

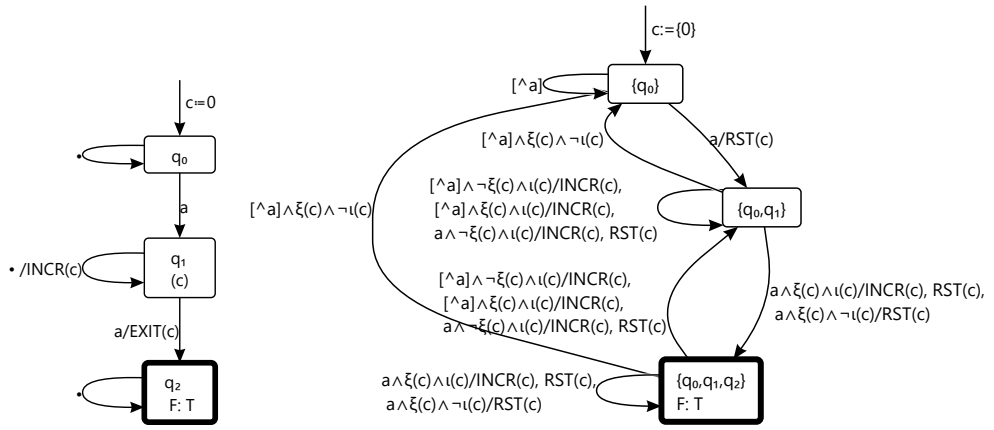
The remaining component is the counting-set operator g. It must summarize the updates of the counter values on transitions of $\Delta_{R,\alpha,\beta}$ as updates of the respective counting sets. The values of counters that are out of scope, hence implicitly zero, will not be tracked in counting sets. Tracking the value of a counter hence starts when A' simulates a transition of A entering the scope of the counter, and ends when no state from the scope is present in the target CsA state.

$$\text{CSOP}(\text{OP}_c, c) \stackrel{\text{def}}{=} \begin{cases} \text{NOOP} & \text{if } \text{OP}_c = \text{NOOP} \wedge p \in \sigma(c) \\ \text{INCR} & \text{if } \text{OP}_c = \text{INCR} \wedge p \in \sigma(c) \\ \text{RST} & \text{if } \text{OP}_c = \text{NOOP} \wedge p \notin \sigma(c) \\ \text{RST1} & \text{if } \text{OP}_c = \text{INCR} \wedge p \notin \sigma(c) \\ \text{RST} & \text{if } \text{OP}_c = \text{EXIT} \\ \text{RST1} & \text{if } \text{OP}_c = \text{EXIT1} \end{cases} \quad (4.18)$$

Let $\Delta_{R,\alpha,\beta}(c)$ be the set of transitions in $\Delta_{R,\alpha,\beta}$ with the target state in the scope of c. The counting-set operator g is built in the form:

$$g(c) \stackrel{\text{def}}{=} \{\text{CSOP}(f(c), c) \mid \exists p, q, \alpha : p \text{-}\{\alpha, f\}\!\mapsto q \in \Delta_{R,\alpha,\beta}(c)\}. \quad (4.19)$$

[5] Recall that the predicates in $\Psi_{\mathbb{C}}$ and $\Psi_{\mathbb{S}}$ are syntactically the same.



(a) $CA(\texttt{.*a.\{4,8\}a})$ (b) Determinisation of the CA into a CsA

Figure 4.3: From a regex via a CA to a deterministic CsA. We are using a notation closely following the formal development. We only use $\text{OP}(c)$ instead of OP_c and abbreviate CANEXIT_c by $\xi(c)$ and CANINCR_c by $\iota(c)$.

Here, $\text{CSOP}(f(c), c)$ denotes the counting-set operation that, given a CA transition $\tau = p \text{-}\{\alpha, f\}\!\!\rightarrow q$, transforms the set of possible values of the counter c at the state p to the set of values obtained at q after taking the transition. It is defined in Equation 4.18 on the right. The set operation induced by the CA transition corresponds to the counter operation on the transition. In the third and fourth case, when the CA transition comes from out of the scope, it is certain that the counter can only have the value 0, which is the same value as produced by EXIT (or EXIT1 when the counter is immediately incremented). The resulting CsA transition is therefore $S \text{-}\{\alpha \wedge \beta, g\}\!\!\rightarrow T$. Note that $g(c)$ ends up empty when the target powerstate is fully out of the scope of c, which semantically corresponds to the implicit reset to $\{0\}$.

Let $r \text{-}\{\alpha, f\}\!\!\rightarrow r' \in \Delta_{R,\alpha,\beta}$. We denote a counting-set operator for $r \text{-}\{\alpha, f\}\!\!\rightarrow r'$ as:

$$f^{cs}(c) = \text{CSOP}(f(c)). \tag{4.20}$$

Observe that A' is deterministic since, for any two distinct transitions $S \text{-}\{\alpha_1, f_1\}\!\!\rightarrow S_1$ and $S \text{-}\{\alpha_2, f_2\}\!\!\rightarrow S_2$, the condition $\alpha_1 \wedge \alpha_2$ is unsatisfiable by virtue of minterms.

Lemma 4.4. *For the CA A and the CsA A' above, we have $L(A') \supseteq L(A)$ and $|Q'| \leq 2^{|Q|}$.*

Proof (idea). We will later provide under additional assumptions a full proof of language equivalence. The language inclusion is proved by showing that the configuration automaton $FA(A')$ of A' simulates $DFA(A)$, more concretely, that each configuration $(R, \mathfrak{s})$ of A', a state of $FA(A')$, simulates the powerstate $(R, \mathfrak{s})$ of $DFA(A)$. The bound on the size of the state space follows from that states of the CsA are sets of states of the CA. The proof of the language inclusion is embedded in the proof of the language equivalence. □

Example 4.4.3. Consider the CA in Figure 4.3a that has states q_0, q_1, and q_2. The state q_0 is initial, the final condition of q_2 is $\top$, and it is $\bot$ for q_0 and q_1. The set of counters is $C = \{c\}$ with $\sigma(c) = \{q_1\}$ (i.e., c is not used and hence implicitly 0 in q_0

and q_2). Finally, $\Sigma = \{\mathtt{a}, [\mathtt{\hat{}a}]\}$. In Figure 4.3a, we compactly represent transitions over all minterms from Σ using '.'. The determinisation starts exploring the CsA from its initial state $S_0 = \{q_0\}$.

Let us focus on the transitions for the input minterm $\alpha = \mathtt{a}$. There are two transitions leaving q_0, namely $\delta_1 = q_0 \text{-}\{\mathtt{a},\textsc{noop}_c\}\!\mapsto q_0$ and $\delta_2 = q_0 \text{-}\{\mathtt{a},\textsc{noop}_c\}\!\mapsto q_1$, both with no guard on c, so we have that $\Gamma_{S_0,\alpha} = \{\top\}$. The guard $\top$ is thus the only choice for the counter minterm β. The set $\Delta_{R,\alpha,\beta}$ of transitions consistent with α and β then contains both $\mathtt{a}$-transitions δ_1 and δ_2 originating from q_0. Since δ_2 is entering the scope of c, it generates the counting-set operation $\textsc{Rst}_c$ according to the third case of Equation 4.18. Since δ_1 stays out of the scope, it does not generate any counting-set operations. We obtain the counting-set operator $g(c) = \{\textsc{Rst}_c\}$ and generate the CsA transition $\tau_1 = \{q_0\} \text{-}\{\mathtt{a}\wedge\beta,\{\textsc{Rst}_c\}\}\!\mapsto \{q_0, q_1\}$.

Next, let us focus on the $\mathtt{a}$-transitions from $S_1 = \{q_0, q_1\}$. Here, $\Gamma_{S_1,\mathtt{a}}$ has the following three satisfiable elements: $\textsc{canExit}_c \wedge \textsc{canIncr}_c$, $\neg\textsc{canExit}_c \wedge \textsc{canIncr}_c$, and $\textsc{canExit}_c \wedge \neg\textsc{canIncr}_c$ (the guard $\neg\textsc{canExit}_c \wedge \neg\textsc{canIncr}_c$ is excluded as it is never satisfied for non-empty sets of positive integers). Let us generate a transition for the second case, $\beta = \neg\textsc{canExit}_c \wedge \textsc{canIncr}_c$. We obtain $\Delta_{S_1,\mathtt{a},\beta} = \{q_0 \text{-}\{\mathtt{a},\textsc{noop}_c\}\!\mapsto q_0$, $q_0 \text{-}\{\mathtt{a},\textsc{noop}_c\}\!\mapsto q_1$, $q_1 \text{-}\{\mathtt{a},\textsc{incr}_c\}\!\mapsto q_1\}$. As before, the first transition does not contribute to g as it stays out of the scope, and the second transition adds $\textsc{Rst}_c$. The third transition adds $\textsc{Incr}_c$ (the second case of Equation 4.18). The resulting CsA transition is thus $\tau_2 = S_1 \text{-}\{\mathtt{a}\wedge\neg\textsc{canExit}_c\wedge\textsc{canIncr}_c,\{\textsc{Incr}_c,\textsc{Rst}_c\}\}\!\mapsto S_1$. The rest of the construction is analogous.

Last, let us also illustrate the simulation of $\mathit{DFA}(A)$ by the constructed CsA transitions. On the word aa, the DFA would execute the run $\{(q_0, c = 0)\} \text{-}\{\mathtt{a}\}\!\mapsto \{(q_0, c = 0), (q_1, c = 0)\} \text{-}\{\mathtt{a}\}\!\mapsto \{(q_0, c = 0), (q_1, c = 0), (q_1, c = 1)\}$. The simulating run of our CsA would start in the initial configuration $\{\{q_0\}, c \in \{0\}\}$. The transition τ_1 would produce the configuration $\{\{q_0, q_1\}, c \in \{0\}\}$ (since $\textsc{Rst}(\{0\}) = \{0\}$) from where τ_2 would produce $\{\{q_0, q_1\}, c \in \{0, 1\}\}$ (since $\textsc{Incr}(\{0\}) = \{1\}$ and $\textsc{Rst}(\{0\}) = \{0\}$). The sequence of configurations precisely encodes the sequence of the DFA powerstates. Indeed, we obtain the sequence of DFA powerstates $(\{q_0\}, c \in \{0\})^{\mathit{DFA}} = \{(q_0, c = 0)\}$; $(\{q_0, q_1\}, c \in \{0\})^{\mathit{DFA}} = \{(q_0, c = 0), (q_1, c = 0)\}$; and $(\{q_0, q_1\}, c \in \{0, 1\})^{\mathit{DFA}} = \{(q_0, c = 0), (q_1, c = 0), (q_1, c = 1)\}$ (recall that q_0 is not in the scope of c hence c has implicitly the value 0 there). □

4.4.4 Uniformity: A Sufficient Semantic Correctness Criterion

Given a CA A, we produce a CsA A' that may overapproximate A in terms of the language. We explain how this may happen and present conditions under which the language stays unchanged. In particular, the overapproximation is caused by non-Cartesian powerstates of $\mathit{DFA}(A)$. (Recall that, in a Cartesian powerstate, states in the scope of a counter must appear with the same set of values of that counter.) A configuration of the CsA cannot encode a non-Cartesian powerstate precisely, it can only overapproximate it. A larger powerstate may then accept a larger language.

Example 4.4.4. Take $R = \mathtt{(a|aa)\{5\}}$ and the $\mathrm{CA}(R)$ shown in Figure 4.4. After reading the word 'aa', $\mathit{DFA}(\mathrm{CA}(R))$ reaches the powerstate $\{(q_0, c = 1), (q_0, c = 2), (q_1, c = 2)\}$, which is not Cartesian because both states are in the scope of the counter c but are paired with different counter values. Our CsA would reach the configuration

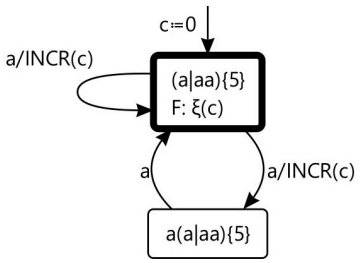


Figure 4.4: $CA(\texttt{(a|aa)\{5\}})$.

$(\{q_0, q_1\}, c \in \{1, 2\})$, which encodes the larger powerstate $\{(q_0, c = 1), (q_0, c = 2), (q_1, c = 1), (q_1, c = 2)\}$ where both states appear with both counter values. □

Uniformity. We now introduce the so-called *uniformity* of a CA as a property under which determinisation preserves the language. Uniformity prevents creation of non-Cartesian powerstates. It includes two conditions.

The first condition prevents the kind of scenario from Example 4.4.4. It requires that for each DFA transition τ', every CA state q in the scope of a counter c within the target DFA state receives the same set of values of c. It requires testing for each such CA state q, that its incoming transitions from which τ' is built induce the same CsA operations for c.

The second condition prohibits two counters being active at once, a scenario which arises from regexes with nested counting. The relation between values of two simultaneously active counters may easily become more intricate then what can be expressed by a Cartesian product of two sets (consider e.g. the regex `a?(a{1}a){2}` and the word `aaa`). The condition requires testing that no state appears in the scope of two counters.

Formally, given a CsA transition $\tau' = S \text{-}\{\alpha \wedge \beta, g\}\!\!\rightarrow T$, a counter c, and a CA state $q \in \sigma(c)$, we define the set $g_q(c)$ of *incoming CsA operations* for c induced by the incoming transitions of q from which τ' is built (α-transitions consistent with β originating in S) as follows:

$$g_q(c) \stackrel{\text{def}}{=} \{\textsc{csop}(f(c), c) \mid \exists p, \alpha : p \text{-}\{\alpha, f\}\!\!\rightarrow q \in \Delta_{S,\alpha,\beta(c)}\} .$$

We call the transition τ' *uniform* iff, for each counter $c \in C$, any two states $q, r \in \sigma(c) \cap T$ have the same sets of incoming CsA operations, i.e., $g_q(c) = g_r(c)$. The CA A is then *uniform* if all transitions of A' are uniform and no state appear in the scope of two counters.

Lemma 4.5. *If an automaton is uniform, then all its states are Cartesian.*

Proof (sketch). By definition CsA, the initial state of CsA is Cartesian (all counters are set to 0). Uniformity implies that the subsequent of a Cartesian configuration is a Cartesian configuration. □

Theorem 7. *If a CA A is uniform, then $L(A) = L(A')$.*

Uniformity can be checked on the fly, while constructing A'. It is also automatically implied when the CA is constructed from certain classes of regexes, as discussed in Section 4.4.5.

Proof of Theorem 7

Let $A = (\mathbb{I}, C, Q, q_0, F, \Delta)$ be a CA. The determinisation algorithm produces the deterministic CsA $A' = (\mathbb{I}, C, Q', S_0, F', \Delta')$. The language of the CsA A' is defined through its underlying configuration FA, $FA(A')$, as $L(A') := L(FA(A'))$. The states of $FA(A')$ are configurations of A'. The result of determisation of $FA(A)$ using the textbook subset construction is $DFA(A)$. In this section, we denote $DFA(A)$ as the automaton $A^d = (\mathbb{I}, C, Q^d, {q_0}^d, F^d, \Delta^d)$ and the configuration automaton $FA(A')$ as $A^c = (\mathbb{I}, C, Q^c, {q_0}^c, F^c, \Delta^c)$.

Theorem 7 can be proven by showing bisimilarity between states of $FA(A')$, A^c, i.e., configurations of the CsA A' and powerstates q^{DFA} of $DFA(A)$, A^d. First, we define what is a bisimulation.

Definition 4.1. *Let $A^1 = (\mathbb{I}, C, Q^1, q_0^1, F^1, \Delta^1)$ and $A^2 = (\mathbb{I}, C, Q^2, q_0^2, F^2, \Delta^2)$ be automata. $\approx$ is a* bisimulation *iff for all $q \in Q^1$, $r \in Q^2$, if $q \approx r$, then the following three conditions hold:*

1. *$\forall q', \alpha, f$: iff $q \text{-}\{\alpha, f\}\!\!\rightarrow q' \in \Delta^1$, then there exists r' such that $r \text{-}\{\alpha, f\}\!\!\rightarrow r' \in \Delta^2$ and $q' \approx r'$,*
2. *$\forall r', \alpha, f$: iff $r \text{-}\{\alpha, f\}\!\!\rightarrow r' \in \Delta^2$, then there exists q' such that $q \text{-}\{\alpha, f\}\!\!\rightarrow q' \in \Delta^1$ and $r' \approx q'$, and*
3. *q is a final state iff r is a final state.*

States q and r are bisimilar *iff there exists a bisimulation $\approx$ such that $q \approx r$.*

It is well-known that bisimulation applies equivalence.

Lemma 4.6. *For automata $A^1 = (\mathbb{I}, C, Q^1, q_0^1, F^1, \Delta^1)$ and $A^2 = (\mathbb{I}, C, Q^2, q_0^2, F^2, \Delta^2)$, $L(A^1) = L(A^2)$ if q_0^1 is bisimilar to q_0^2.*

Let us recall that the powerstates of A^d encode the states of A^c, as mentioned in Section 4.4.2. The encoding defines the bisimulation relation. The configurations $(R, \mathfrak{s})$ of CsA A^c, where R is a set of states of A and $\mathfrak{s}$ is a counting-set memory, are decoded as the set $(R, \mathfrak{s})^{DFA}$ of pairs $\{(r_i, \mathfrak{m}_i)\}_{1 \leq i \leq n}$ consisting of a state $r_i \in R$ and a counter memory $\mathfrak{m}_i$ such that for all $c \in C$, $\mathfrak{m}(c) \in \mathfrak{s}(c)$ if $c \in \sigma(r)$, alse $\mathfrak{m}(c) = 0$.

A state $U = \{(r_i, \mathfrak{m}_i)\}_{1 \leq i \leq n}$ is encoded as the CsA configuration

$$enc(U) = (\bar{R}, \bar{\mathfrak{s}})$$

where $\bar{\mathfrak{s}}(c) = \{\mathfrak{m}_i(c) \mid q_i \in \sigma(c) \wedge (r_i, \mathfrak{m}_i) \in U\}$ and $\bar{R} = \{r_i \mid (r_i, \mathfrak{m}_i) \in U\}$.

In general, the encoding is not injective (it is ambiguous). A set of number-interpretations appearing with the state q is broken down to sets of values of individual counters, as also explained in Section 4.4.4. When restricted to Cartesian DFA states, enc is injective and decoding is its inversion.

We define the relation of encoding, $\prec_{cod} \subseteq Q^d \times Q^c$, so that for $U \in Q^d$ and $T \in Q^c$, $U \prec_{cod} T$, iff $enc(U) = T$.

Now we prove using Lemma 4.6 that $L(A^d) = L(A^c)$ by proving that:

a) ${q_0}^d \prec_{cod} {q_0}^c$, and

b) $\prec_{cod}$ is bisimulation.

Showing (a) is straightforward from the definition of the initial configurations of the automata. We can show easily that ${q_0}^c = \{(q_0, \{c \mapsto \{0\}\}_{c \in C})\}$ and $enc({q_0}^c) = (\{q_0\}, \{\, c \mapsto \{0\} \mid c \in C \,\})$ which is obviously the initial state of A^d.

To show (b) we have to prove that:

1. For all $T \in Q^c$ and α-transitions $T \text{-}\{\alpha\}\!\!\rightarrow T'$ in Δ^c, if $T \prec_{cod} U$ where $U \in Q^d$, there is $U \text{-}\{\alpha\}\!\!\rightarrow U'$ in Δ^d such that $T' \prec_{cod} U'$.

2. For all $U \in Q^d$ and α-transitions $U \text{-}\{\alpha\}\!\!\rightarrow U'$ in Δ^d, if $U \prec_{cod} T$, there is $T \text{-}\{\alpha\}\!\!\rightarrow T'$ in Δ^c such that $U' \prec_{cod} T'$.

3. T is a final state iff U is a final state.

Let U be a state of Q^d and a symbol $\alpha \in \Sigma$. The transitions of A^d are built from the set of α-transitions of CA A leading from U defined as:

$$\Delta_{\alpha,U} = \{r \text{-}\{\alpha,f\}\!\!\rightarrow r' \in \Delta \mid \exists \mathfrak{m} : \mathfrak{m} \models \varphi_f \wedge (r, \mathfrak{m}) \in U\}. \tag{4.21}$$

Let $T = (R, \mathfrak{s})$ and $T' = (R', \mathfrak{s}')$ be states of Q^c and $T \text{-}\{\alpha\}\!\!\rightarrow T' \in \Delta^c$. Let us recall that the transition is build from a transition of CsA, $R \text{-}\{\alpha \wedge \beta, g\}\!\!\rightarrow R'$, which is constructed from a set:

$$\Delta_{R,\alpha,\beta} = \{r \text{-}\{\alpha,f\}\!\!\rightarrow r' \in \Delta \mid r \in R, \textbf{\textit{Sat}}(\varphi_f \wedge \beta)\},$$

where, according to Equation 4.17, β represents a minterm from the set of minterms:

$$\Gamma_{R,\alpha} = \mathit{Minterms}(\{grd(\mathrm{OP}_c) \mid r \text{-}\{\alpha,f\}\!\!\rightarrow r' \in \Delta, r \in R \wedge c \in \sigma(r), \mathrm{OP}_c \in f\}),$$

defined in Equation 4.16.

Lemma 4.7. *Let $T = (R, \mathfrak{s})$ be a state of Q^c, $\alpha \in \Sigma$ and $\beta \in \Gamma_{R,\alpha}$. The α-successor of T in A^c is:*

$$T' = (R', \mathfrak{s}'),$$

where

$$R' = \{r' \mid r \text{-}\{\alpha,f\}\!\!\rightarrow r' \in \Delta_{R,\alpha,\beta}\} \tag{4.22}$$

and

$$\mathfrak{s}' = \bigcup_{r \text{-}\{\alpha,f\}\rightarrow r' \in \Delta_{R,\alpha,\beta}} f^{cs}(\mathfrak{s}). \tag{4.23}$$

The union is computed componentwise, $\forall c \in C, \forall \mathfrak{s}_1, \mathfrak{s}_2 : \mathfrak{s}_1 \cup \mathfrak{s}_2(c) = \mathfrak{s}_1(c) \cup \mathfrak{s}_2(c)$.

Proof. It follows from the definition of Equation 4.17. □

Lemma 4.8. *For some $R \subseteq Q$, $\alpha \in \Sigma$, $\beta \in \Gamma_{R,\alpha}$ and a transition $r \text{-}\{\alpha,f\}\!\!\rightarrow r' \in \Delta_{R,\alpha,\beta}$:*

$$\mathit{csop}(f(c), c)(\mathfrak{s}(c)) = \{f(c)(k) \mid k \in \mathfrak{s}(c) \wedge k \models \varphi_f^c\}.$$

Proof. It follows from the definition of CSOP 4.18 and semantic of CA and CsA. □

Lemma 4.9. *Let A^d be a Cartesian DFA, $U \in Q^d$ and $T = (R, \mathfrak{s})$ be a state of Q^c such that $T \prec_{cod} U$. Let $\alpha \in \Sigma$ and $\beta \in \Gamma_{R,\alpha}$. Then $\Delta_{R,\alpha,\beta} = \Delta_{\alpha,U}$.*

Proof. Let us recall that β is a conjunction of counter guards φ on the α-transitions τ originating in R or their negations. Let $\tau = r\text{-}\{\alpha,f\}\!\!\rightarrow r'$ be a transition of Δ. Since $\mathfrak{s} \models \beta$, it holds that $\mathfrak{s} \models \varphi_f$ for $\tau \in \Delta_{R,\alpha,\beta}$ and $\mathfrak{s} \models \neg\varphi_f$ for $\tau \notin \Delta_{R,\alpha,\beta}$.

Let $\tau \in \Delta_{R,\alpha,\beta}$. Therefore, for each counter $c \in C$ there is a value $k_c \in \mathfrak{s}(c)$ such that $k_c \models \varphi_f^c$ for a guard φ_f appearing in a conjunction β and $k_c \models \neg\varphi_f^c$ for all guards φ_f such that their negation appears in β. Let $\mathfrak{m}$ be a memory such that for each counter $c \in C$, $\mathfrak{m}(c) = k_c$ if $c \in \sigma(r)$, else $\mathfrak{m}(c) = 0$. From the definition of decoding, since it creates a Cartesian DFA state, it holds that there is a configuration $(r, \mathfrak{m}) \in U$. By the definition of $\mathfrak{m}$, τ is enabled in $(r, \mathfrak{m})$ (because $\mathfrak{m} \models \varphi_f$). Therefore, $\tau \in \Delta_{\alpha,U}$.

Let $\tau \notin \Delta_{R,\alpha,\beta}$. Therefore, there is a counter $c \in C$ such that for all $k_c \in \mathfrak{s}(c)$, $k_c \not\models \varphi_f^c$ for a guard φ_f appearing in a conjunction β or $k_c \not\models \neg\varphi_f^c$ for at least one guard φ_f such that its negation appears in β. From the definition of decoding, it holds that there is no $(r, \mathfrak{m}) \in U$ such that $\mathfrak{m} \models \varphi_f$. Therefore, τ is not enabled in any $(r, \mathfrak{m}) \in U$ and hence, $\tau \notin \Delta_{\alpha,U}$. □

Now, we finally prove that $\prec_{cod}$ is bisimulation. Let $T = (R, \mathfrak{s})$ and $T' = (R', \mathfrak{s}')$ be states of Q^c, A^d be a Cartesian DFA, $U \in Q^d$, $\alpha \in \Sigma$ and $\beta \in \Gamma_{R,\alpha}$. We will first prove the point (1) of the definition of the bisimulation, that is, that for all $U \in Q^d$ and an α-transition $U\text{-}\{\alpha\}\!\!\rightarrow U'$ in Δ^d, if $U \prec_{cod} T$, that is, $enc(U) = T$, there is $T\text{-}\{\alpha\}\!\!\rightarrow T' \in \Delta^c$ such that $U' \prec_{cod} T'$, that is, $enc(U') = (\bar{R}', \bar{\mathfrak{s}}') = T'$.

By Lemma 4.7, we have that $R' = \{r' \mid r\text{-}\{\alpha,f\}\!\!\rightarrow r' \in \Delta_{R,\alpha,\beta}\}$ and $\mathfrak{s}' = \bigcup_{r\text{-}\{\alpha,f\}\rightarrow r' \in \Delta_{R,\alpha,\beta}} f^{cs}(\mathfrak{s})$. By the definition of encoding, we have that $\bar{R}' = \{\, r' \mid r\text{-}\{\alpha,f\}\!\!\rightarrow r' \in \Delta_{\alpha,U} \,\}$. By Lemma 4.9, $R' = \{\, r' \mid r\text{-}\{\alpha,f\}\!\!\rightarrow r' \in \Delta_{R,\alpha,\beta}\} = \bar{R}' = \{\, r' \mid r\text{-}\{\alpha,f\}\!\!\rightarrow r' \in \Delta_{\alpha,U} \,\}$. It remains to be shown that $\mathfrak{s}' = \bar{\mathfrak{s}}'$.

To do that, because of 4.23, we have to show that for all $c \in C$, $\bigcup_{r\text{-}\{\alpha,f\}\rightarrow r' \in \Delta_{R,\alpha,\beta}} f^{cs}(c)(\mathfrak{s}(c)) = \bar{\mathfrak{s}}'(c)$. Let us fix $c \in C$. First, we will show that if there is $\tau = r\text{-}\{\alpha,f\}\!\!\rightarrow r' \in \Delta_{R,\alpha,\beta}$ such that $n' \in f^{cs}(c)(\mathfrak{s}(c))$, then there is $n' \in \bar{\mathfrak{s}}'(c)$. Lemma 4.8 and the definition of f^{cs} implies that:

$$f^{cs}(c)(\mathfrak{s}(c)) = \textsc{csop}(f(c), c)(\mathfrak{s}(c)) = \{f(c)(k) \mid k \in \mathfrak{s}(c) \wedge k \models \varphi_f^c\}. \qquad (4.24)$$

Therefore, since $n' \in f^{cs}(c)(\mathfrak{s}(c))$, there must be some $n \in \mathfrak{s}(c)$ such that $n \models \varphi_f^c$ and applying $f(c)$ on n we get n', that is, $f(c)(n) = n'$. From the definition of encoding, if there is $n \in \mathfrak{s}(c)$, there must exist some $(r, \mathfrak{m}) \in U$ such that $\mathfrak{m}(c) = n$. Because $\tau \in \Delta_{R,\alpha,\beta}$, from Lemma 4.9, we have that $\tau \in \Delta_{\alpha,U}$. By the definition of $\Delta_{\alpha,U}$, since $\tau \in \Delta_{\alpha,U}$ there must be some $(r, \mathfrak{m}_1) \in U$ where $\mathfrak{m}_1 \models \varphi_f$. Since A^c is uniform, we know, from Lemma 4.5, that all its states are Cartesian. Therefore, we know that there is some $(r, \mathfrak{m}_2) \in U$ where $\mathfrak{m}_2$ is the same as $\mathfrak{m}_1$ except for $\mathfrak{m}_2(c) = n$. From the definition of CA, since φ_f is a conjuction of unary predicates over counters, then $\mathfrak{m}_2 \models \varphi_f$. Since $\mathfrak{m}_2(c) = n$ and $f(c)(n) = n'$, it means that $f(c)(\mathfrak{m}_2(c)) = n'$. Hence, by Equation 4.24, $n' \in \bar{\mathfrak{s}}'(c)$.

Second, we will show that if there is $n' \in \bar{\mathfrak{s}}'(c)$, then there is $\tau = r\text{-}\{\alpha,f\}\!\!\rightarrow r' \in \Delta_{R,\alpha,\beta}$ such that $n' \in f^{cs}(c)(\mathfrak{s}(c))$. By the definition of $\Delta_{\alpha,U}$, since $n' \in \bar{\mathfrak{s}}'(c)$, there must be some $\tau = r\text{-}\{\alpha,f\}\!\!\rightarrow r' \in \Delta_{\alpha,U}$ such that $f(c)(\mathfrak{m}(c)) = n'$. Since $\tau \in \Delta_{\alpha,U}$, then there must be some $(r, \mathfrak{m}) \in U$ such that $\mathfrak{m} \models \varphi_f$ and $\mathfrak{m}(c) = n$ such that $f(c)(n) = n'$. By the definition of decoding, if there is $(r, \mathfrak{m}) \in U$, then the must be $n \in \mathfrak{s}(c)$. Because $\tau \in \Delta_{\alpha,U}$, from Lemma 4.9, we have that $\tau \in \Delta_{R,\alpha,\beta}$. Since $n \in \mathfrak{s}(c)$, $n \models \varphi_f^c$ and applying $f(c)$ on n we get n', then there must be $n' \in f^{cs}(c)(\mathfrak{s}(c))$, that is $f(c)(n) = n'$.

Now we prove the point (2) of the definition of the bisimulation, that is, that for all $T \in Q^c$ and an α-transition $T \dashv\{\alpha\}\mapsto T'$ in A^c, if $T \prec_{cod} U$, there is $U \dashv\{\alpha\}\mapsto U'$ in Δ^d such that $T' \prec_{cod} U'$. The argument is analogous and very similar to point (1).

Finally, we prove the point (3) of the definition of the bisimulation, that is, that U is a final state iff T is a final state. U is final iff $(r, \mathfrak{m}) \in U$ such that $\mathfrak{m} \models F(r)$. T is final iff $\mathfrak{s} \models \bigvee_{q \in R} F(q)$. Therefore, by the definition of encoding, it holds that $\mathfrak{m} \models F(r)$ iff $\mathfrak{s} \models F(r)$ and hence T is final.

This concludes the proof of Theorem 7.

4.4.5 Syntactic Correctness Criteria

Uniformity is only a semantic property. Below, we show examples of actual regexes that do and do not lead to uniform CAs and discuss some simple syntactic classes of regexes that imply uniformity. A detailed study of syntactic classes of regexes that guarantee uniformity is, however, beyond the scope of this thesis and a part of our future work.

The regexes that induce non-uniform CAs are often those where, intuitively, there is a position in some input text that may either be matched against the first character of a counted sub-expression or against some inner character of the same sub-expression. In such a situation, there may be two runs of the induced CA: one that increments the associated counter (the increment happens) at that position and moves to some state q, and the other that leaves the counter as it is, while in its scope, and moves into a different state r. The counter value then depends on the state: it is different in q and in r. The corresponding DFA state is then non-Cartesian and the CA is non-uniform.

Example 4.4.5. We present several commented examples of regexes with non-uniform CAs where our determinisation overapproximates the language of the obtained CsA.

- `(a|ab|ba){5}` — the string 'aba' could be matched as `a` followed by `ba`, having incremented the counter twice, or as `ab` that is followed by the prefix `a` of `ab`, having incremented the counter once only.

- `.*(aa){6}` — assuming a sequence of a's on the input, the counter may be either incremented on odd characters and left unchanged on even ones, or the other way around. As the counter values depend on the position within the `aa` (and hence on the CA state), the CA cannot be uniform. Note that the prefix `.*` is quite usual as it corresponds to searching for the regex `(aa){6}` anywhere in the input string. Nested counting is often problematic, however, many of such examples may still be solved quite efficiently by unfolding one of the counters.

- `.*(a{2}){2}` — after reading 'aa', if the value of the outer counter is 1, then the value of the inner counter must be 0. This is a non-trivial relation between the values of the two counters, which is not Cartesian.

- `.*.(ab){2}` — assuming a sequence of a's (exactly n+1 of then) on the input, the counter may be either incremented reading the first a from (ab) moving from q to r, or set to 0 reading a symbol from `.` before the brackets entering q. Therefore, we can be in q with increasing value of the counter, even in the original CA is set to 0. □

Table 4.1: Statistics for the graphs in Section 4.5 (times are given in seconds). For Chipmunk, we provide several times: "total" is the total time, "CA" is the time for translating a regex into a (nondeterministic) CA, "CsA" is the time of determinisation of the CA into a CsA, and "match" is the time spent when matching the input text.

	RE2	grep	.NET	SRM	Chipmunk			
					total	CA	CsA	match
mean	36.11	34.38	9.12	26.78	1.73	0.05	0.23	0.69
median	0.10	0.70	0.76	0.73	1.03	0.03	0.04	0.68
std. dev	157.05	147.17	52.10	106.16	7.27	0.29	2.73	0.29
timeouts	1	11	8	16	0			

Syntactic classes of regexes that guarantee uniformity. The syntactic class of regexes that guarantees uniformity includes monadic regexes (since there is only one state in the scope of a counter) and can be extended beyond the monadic regexes. Here we give an example of another class that covers all the above mentioned examples. The regexes of this class are of the form:

$$\alpha_0(\alpha_1 \dots \alpha_n)\{\ell, k\} \text{ or } \hat{}(\alpha_1 \dots \alpha_n)\{\ell, k\}$$

$$\text{s.t. } [\![\alpha_0]\!] \text{ and } [\![\alpha_1]\!] \text{ is disjoint from every } [\![\alpha_i]\!],\ 1 < i \leq n.$$

Intuitively, the disjointness with α_1 ensures that the generated CA will only be able to process α_1 through an increment transition at the beginning of a new iteration of the loop, with no possibility of having a conflicting NOOP transition that could read the same symbol inside the body of the counting loop (which is exactly what happens with the second symbol a in Example 4.4.4). The condition of disjointness of α_0 is important in the last example of Example 4.4.5.

Nonetheless, the class of regexes where our determinisation preserves the language seems to be much larger. For instance, it includes the regexes `((aa)|(bb))*aa((aa)|(bb))`$\{k\}$ and `.*((Natasha)|(Yurij)|(banza{8}j!))`$\{5\}$, despite that the latter even uses nested counting.

4.5 Experimental Evaluation

We have implemented the approach described in the previous sections in a C# prototype called Chipmunk available at [99] and evaluated its pattern matching capabilities against other state-of-the-art regex matchers on patterns with bounded repetition. We focused on comparison against Google's RE2 library [42][6], an automata-based matcher designed to be fast, predictable, and resilient against ReDoS attacks. We also include other three efficient matchers into the comparison, namely the standard GNU grep program [32] (version 3.3), the .NET standard library regex matcher from System.Text.RegularExpressions [65], and Symbolic Regex Matcher (SRM) [82].

[6] We used the version 2019-01-01 of RE2 via the command line interface re2g from https://github.com/akamai/re2g.

We run our benchmarks on a machine with the Intel(R) Xeon(R) CPU E3-1240 v3 @ 3.40 GHz running Debian GNU/Linux (we use the Mono platform [75] to run .NET tools). To avoid issues with generating exact matches, which might differ for different tools, the tools were run in the setting where they counted the number of lines matching[7] the given regex (e.g. the `-c` flag of `grep`).

4.5.1 ReDoS Resiliency

Our main experiment focuses on the resilience of the regex matching engines against ReDoS attacks. The regexes used for this experiment were selected (1) from the database of over 500,000 real-world regexes coming from an Internet-wide analysis of regexes collected from over 190,000 software projects [27]; (2) from databases of regexes used by *network intrusion detection systems* (NIDSes), in particular, Snort [63], Bro [80], Sagan [95], and, moreover, the academic papers [110, 102]; (4) the RegExLib database of regexes [78]; and (5) industrial regexes from Chapter 5, originally used for security purposes. From these, we created our set of benchmarks by the following steps:

(1) We selected regexes that contained counting loops whose sum of upper bounds was larger than 20. This let us focus on regexes where the use of counting makes sense (there are surprisingly many regexes occurring in practice where the use of a counting loop is unnecessary, e.g., regexes containing sub-expressions similar to `a{0,1}` or even just `a{1}`). Moreover, we also removed all except 26 regexes with counters bigger than 1,000, which cannot be handled by `RE2`. We left the 26 regexes as representatives of "large" counters. This left us with 5,000 regexes.

(2) Then, we selected regexes R such that $\mathrm{CA}(R)$ was uniform (cf. 4.4.4), i.e., the CsA produced by our algorithm was precise. After this step, a vast majority, 4,429 of the regexes, remained.

(3) For the regexes that remained, we used a lightweight ReDoS generator designed to exploit counting (cf. 4.5.3) to generate ~10 MiB long input texts. In particular, we managed to generate "adversarial" input texts for 1,789 regexes (for the rest of the regexes, either the underlying state space was too small, so the generator could not construct the text, or the generation hit the timeout of 600 s). Our benchmark data set is available at [50].

We ran all tools on the generated benchmarks (counting the number of lines of the input text matching the regex) and give scatter plots comparing the running times of the tools in Figure 4.5 and Figure 4.6 (the timeout was 600 s). On the bottom and the left-hand side of every plot, there are rug plots illustrating the distribution of the data points. Note that the axes are logarithmic, so the difference between data points grows as these points are away from zero (in particular, differences of values smaller than 1 s are negligible). The semantics of regexes supported by `grep` differs from the one supported by other tools, so we only considered the cases when the number of matches was the same when comparing with `grep`). In the plots, the data points between

[7] We consider the standard semantics of "matching" used by `grep`, i.e., a line matches a regex R if it contains a string that is in $L(R)$, unless it contains start-of-line (`^`) or end-of-line (`$`) anchors, in which case the matched string needs to occur at the start and/or at the end of the line respectively.

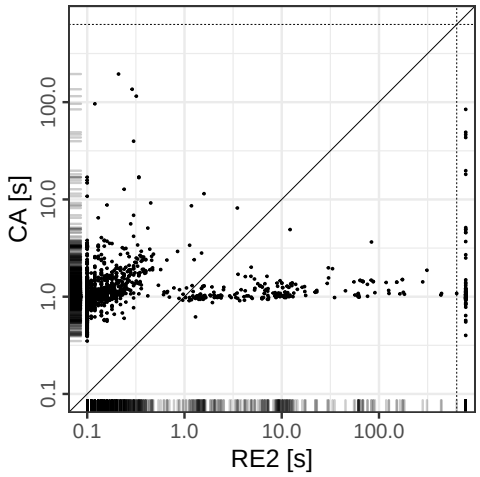


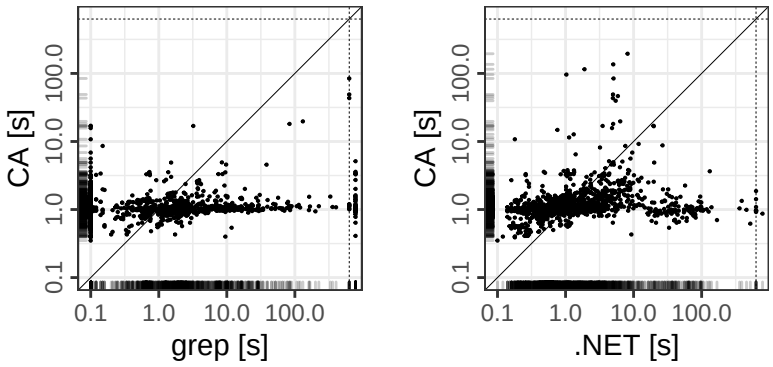
Figure 4.5: The comparison of running times of Chipmunk and RE2 on our benchmark set (Chipmunk wins: 287/1,789). Note that the axes are logarithmic, the dashed lines denote the timeout (600 s), and the data points between the dashed lines and the edge of a plot represent benchmarks where the tool did not run successfully.

the dashed lines and edges of the plots represent errors, e.g. due to the regex being rejected (for counters >1,000 for RE2) or being interpreted using a different semantics (in the case of grep).

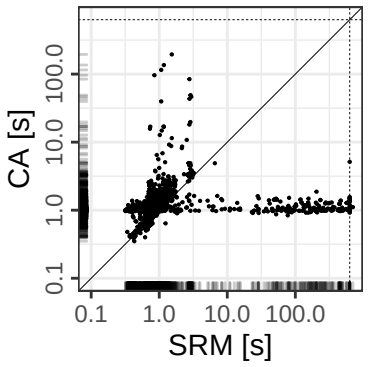
In Figure 4.5, we compare Chipmunk with RE2. We wish to point out the following interesting observations. Although RE2 wins more often on the whole benchmark set (our prototype does not include the many advanced optimizations present in RE2), there is a number of benchmarks (287) where its performance significantly deteriorates, and Chipmunk is faster. In particular, there are 89 benchmarks where the time of RE2 is bigger than 10 s, i.e., its speed drops below 1 MiB/s (we consider this speed of processing denotes a successful ReDoS attack, even though the limit may be significantly larger in practice[8]). For Chipmunk, the number of benchmarks that took over 10 s was only 22; in fact, all except 3 benchmarks finished within 100 s—the blow-up in these 3 benchmarks is not caused by the counters but rather by many '|' and '?' operators, so over 70 % of the total time is spent by constructing the CsA. If used, e.g., in an NIDS, the CsA would be created only once and then used for matching giga-/terabytes of data, so the initial overhead could be neglected.

Comparing with the other tools (Figure 4.6) and also clearly visible in the corresponding rug plots and the statistics in Table 4.1, we can observe that the performance of Chipmunk is much more robust than the performance of the other tools; the mean time and standard deviation of Chipmunk is significantly lower than the rest of the tools. In particular, from the benchmarks where Chipmunk was faster than RE2, the time of Chipmunk on all except two benchmarks was almost the same (including them, the standard deviation was 0.37). We provide four times for Chipmunk: "total": the total user time of matching (measured using the GNU time utility), "CA": the time for translating

[8]The required processing speed depends on the application. NIDSes performing deep packet inspection may require a line-processing speed of units or tens of GiB/s [102], while application servers validating user inputs may suffice with units or tens of MiB/s.



(a) Chipmunk wins: 862/1,425 (b) Chipmunk wins: 708/1,789



(c) Chipmunk wins: 345/1,789

Figure 4.6: The comparison of running times of Chipmunk with grep, .NET, and SRM on our benchmark set. Note, that the axes are logarithmic, the dashed lines denote the timeout (600 s), and the data points between the dashed lines and the edge of a plot represent benchmarks where the tool did not run successfully.

the input regex into a CA, "CsA": the time it took to determinise the CA into a CsA, and "match": the time of matching the input text with the CsA. Note that, in the tables, there is a noticeable discrepancy between the sum "CA" + "CsA" + "match" and "total", which is due to the .NET Framework overhead, such as just-in-time compilation and (in particular) the garbage collector.

In Table 4.7, we give a selection of interesting benchmarks. These contain benchmarks that are difficult for usually more than one tool. We emphasize the benchmarks coming from the NIDSes Snort and Bro. Notice that, for most of them, matching using RE2 (and also other tools) gets extremely slow. Slow matching over these regexes can have disastrous consequences for network security, potentially completely eliminating a given NIDS.

The CsAs produced by Chipmunk were also much smaller than the corresponding DFAs. The CsAs have on average 29 states (median: 7) and 306 transitions (median: 11). On the other hand, classical NFAs constructed from the regexes have on average 112 states (median: 52), and when determinised, the resulting DFAs have on average 2,802 states (median: 67) and 10,384 transitions (median: 107). Using CsAs significantly lowers the chance that determinisation explodes.

Figure 4.7: Selection of interesting benchmarks. 'TO' denotes a timeout (600 s) and '—' denotes an error. Due to space constraints, in the 'Regex' column, '. . . ' denotes omitted parts of the regexes (we tried to preserve the parts containing occurrences of the counter operator) and '∼' denotes breaking a regex into two lines.

Source	Regex	RE2	grep	.NET	SRM	CA			
						total	CA	CsA	match
Snort	`.*[aA][uU][tT][hH]...[iI][cC] ∼ ∼[^\x0A]{512}`	11.27	7.8	361.1	555.56	1.04	0.03	0.05	0.31
Snort	`\x20[^\x21\x22]{500}`	439.98	0.11	2.20	TO	1.08	0.03	0.04	0.83
Snort	`^RCPT TO\x20\s*[\w\s@\.]{200,}∼ ∼\x20[\w\s@\.]{200,}...`	340.7	—	TO	TO	1.68	0.03	0.07	0.89
Snort	`php.*\x20[^\n]{256}`	176.75	0.10	1.22	TO	1.08	0.04	0.07	0.74
Snort	`^(NT|CallBack|SID|TimeOut)\s*∼ ∼\x20\s*[^\n]{512}`	164.11	0.12	14.59	229.41	1.07	0.03	0.07	0.72
Snort	`.*[nN][eE][wW]... [^\x20]{100}`	0.13	1.26	39.92	0.74	0.81	0.03	0.04	0.65
Bro	`^[nN][aA][mM][eE]=s*[^\r\n\x3b∼ ∼\x20\x09\x0b\x2c]{300}`	128.57	12.24	0.51	76.48	1.15	0.03	0.04	0.94
[21]	`_.{39}`	22.96	225.34	1.94	357.68	1.12	0.03	0.04	0.79
[21]	`(.{1,980}[,])\s+(\S)`	260.59	TO	308.66	0.63	1.07	0.03	0.05	0.59
[21]	`(_a ){64999}_a`	—	—	TO	TO	0.96	0.03	0.04	0.51
[21]	`\[{50000}a\]{50000}`	—	—	4.36	TO	5.13	0.02	0.02	0.41
[21]	`^QS([NDR])(.{4})(.{6})(\d{8})...∼ ∼(.{4})(.{6})(.{8})(.{8})(.)$`	0.12	0.10	1.03	0.85	96.20	0.04	81.64	0.65

The effect of nondeterministic counting. We say that a regex contains *nondeterministic counting* if, when translated into a CA A using the algorithm in Section 4.3, there is a word w such that A can over w reach two configurations with different values of some counter.

Regexes with nondeterministic counting are the main focus of our benchmark. Namely, they constitute 67 % of the 1,789 regexes used. From the 1,284 regexes that were at least *slightly* problematic for some of the other tools except `Chipmunk` (it took some tool ≥ 1 s), 73 % of them were with nondeterministic counting. From the 454 regexes that were *significantly* problematic for some of the other tools (it took some tool ≥ 10 s), 85 % of them had nondeterministic counting. From the 109 regexes that were *problematic* for *all* other tools (≥ 1 s), 100 % were with nondeterministic counting. As shown in the results above, our approach can deal with nondeterministic counting quite well.

Adversarial regexes. Another ReDoS scenario is when the attacker can control the regex to be used for matching. Creating a counting regex causing efficiency problems for a given text is easier than generating adversarial texts. For instance, the regex `[a-zA-Z().,' ]*[a-zA-Z ] [a-zA-Z();' ]{250}` was obtained as a modification of the running example `.*a.{`k`}` (where `a` appears k positions from the end). When run on a ∼4 MiB English text with sufficiently long lines, `RE2` took 86 s, `grep` took 26 s, while `Chipmunk` took only 1.1 s. Similar examples could be obtained from regexes from Section 4.5.1 for which some specific difficult text can be generated, namely by widening their character classes. Our approach solves a large class of the dangerous cases, allowing one to significantly alleviate restrictions put on the user for security/efficiency reasons.

We admit that similarly difficult regexes could be crafted to fall out from the fragment we handle: they could for instance use nesting of counters. Nevertheless, our approach

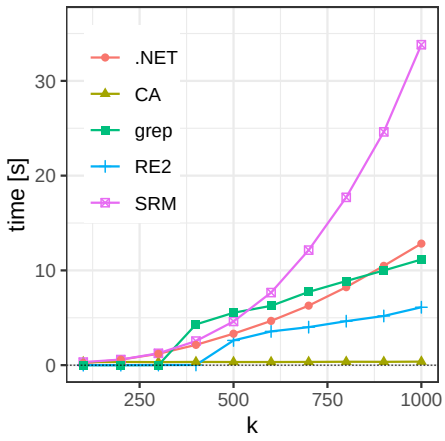


Figure 4.8: Running times of the tools on the regex `(_a ){k}_a` where k is a parameter.

still solves a large class of such dangerous cases, allowing to significantly alleviate restrictions put on the user from security/efficiency reasons, and improve guarantees. RE2 currently allows counting up to 1000, grep only up to 250, and this still does not exclude harmful uses of counting, as we showed above. On the contrary, when using our approach, anything withing the fragments described in Section 4.4.4 can be allowed with any counter bounds and with strong performance guarantees.

4.5.2 Robustness wrt Counter Values

In our next experiment, we measured the ability of the tools to cope with increasing counter bounds. For this, we selected the regex `(_a ){`k`}_a` where k is a parameter (the original regex `(_a ){64999}_a` comes from [27]) and measured the time the tools took on a ~500 KiB text created by our generator for increasing values of k. We give the results in Figure 4.8.

With the increasing value of k, the time needed by `Chipmunk` stays constant, around 0.35 s, while the time needed by other tools grows. In particular, `.NET` and `SRM` have cubic trends wrt the value of k, while `RE2` and `grep` grow linearly. Notice that, for `RE2` and `grep`, their matching time is low (around 0.01 s) until they reach a threshold from which they start behaving linearly. This corresponds to the situation when the size of the cache for storing states of the NFA-to-DFA construction is not enough to accommodate the DFA states exercised by the input adversarial text. This yields repeated flushing of the cache, making it ineffectual.

4.5.3 Adversarial Text Generation

`RE2` and `grep` store powerstates of the NFA-to-DFA construction in a cache. In typical cases, the amount of cache misses is low and almost the entire text is processed using the cache, which is extremely fast. If the cache, however, exceeds a given size, it is flushed. If the input text is such that the DFA run sees many different states, then cache misses are frequent, so large powerstates need to be constructed often, and the performance of the matching drops.

Therefore, we focus on generating texts that force exploration of many new large powerstates. In essence, we explore the configuration space of the CsA with the goal of finding as many large configurations as possible, with the focus on generating large counting sets. We partially drive the search towards loops in the CsA structure that have a potential to create large counting sets: the loops use counters with large bounds, do not contain exits, and contain RST or RST1 operations. We will give the details and elaborate on this in Chapter 5.

4.5.4 A Note on the Maturity of the Tools

The aim of our experiments is comparing algorithms rather than tools, and it should be noted that `Chipmunk` is much less optimized than the rest. This holds especially for `RE2` and `grep`, which have both been actively developed for over 10 years and the amount of engineering effort invested into making them fast is substantial. The optimizations are both high-level, such as using the Boyer-Moore algorithm for skipping sections of the input text, and low-level, such as using C/C++, on-the-fly determinisation, or optimizing memory accesses [22, 46]. On the other hand, although there have been some optimizations done in `Chipmunk` (such as finding a start of a match), their nature is still quite simple. The three tools are, however, all based on the same principle of using deterministic automata, and many of the optimizations and heuristics in `RE2` and `grep` (at least all of those mentioned above) could be directly re-applied in our setting. `SRM` builds on the .NET framework and reuses the `.NET` regex parser while replacing the built-in backtracking back-end matcher with a matching engine based on Brzozowski-style symbolic derivatives to create the DFA on the fly. In fact, `Chipmunk` builds on the open-source codebase of `SRM` and extends it with counters.

4.6 Implementation

We discuss some important implementation details of our matcher `Chipmunk` relating to the implementations of $\mathbb{I}$, $\mathbb{C}$, $\mathbb{S}$, and the product algebra $\mathbb{I}\times\mathbb{S}$, and their role in the CA determinization algorithm. All algorithms are implemented in C# using the open source Microsoft.Automata library [67].

BDD algebra. We use an effective Boolean algebra $\mathcal{B}$ that we call a *BDD algebra* with universe $\mathbb{N}$. The basic predicates of $\mathcal{B}$ have the form β_i where $i \geq 0$ is a bit position and $[[\beta_i]]_{\mathcal{B}}$ is the set of all n whose i'th bit in binary is 1. $\Psi_{\mathcal{B}}$ consists of BDDs and its Boolean operations are corresponding BDD operations. We write $|$ for $\vee_{\mathcal{B}}$, $\&$ for $\wedge_{\mathcal{B}}$, and $\overline{\psi}$ for $\neg_{\mathcal{B}}\psi$.

Algebras $\mathbb{C}$ and $\mathbb{S}$. We use $\mathcal{B}$ to encode predicates over a given collection of counters c_i for $0 \leq i \leq k$ as follows. We represent the condition $\textsc{canExit}_i$ with the $\mathcal{B}$-predicate β_{2i} and the condition $\textsc{canIncr}_i$ with the $\mathcal{B}$-predicate β_{2i+1}. Thus, assuming c_i is in its valid range, in $\mathbb{C}$,

- $\beta_{2i}\&\beta_{2i+1}$ represents the case $\boldsymbol{min}_i \leq c_i < \boldsymbol{max}_i$,
- $\beta_{2i}\&\overline{\beta_{2i+1}}$ represents the case $c_i = \boldsymbol{max}_i$,
- $\overline{\beta_{2i}}\&\beta_{2i+1}$ represents the case $c_i < \boldsymbol{min}_i$.

The predicate $\overline{\beta_{2i}}\&\overline{\beta_{2i+1}}$ represents the impossible condition that $c_i < \boldsymbol{min}_i$ and $\boldsymbol{max}_i \leq c_i$. Therefore, the predicate $\beta_{2i}|\beta_{2i+1}$ is used to elimiate the impossible case and is always asserted in conjunction with any counter predicate for c_i. Analogously for $\mathbb{S}$ because we work with nonempty sets.

Input algebra $\mathbb{I}$. Given a collection of character classes, obtained from a regex R, we first compute all the minterms over those character classes. Let the minterms be $\Sigma = \{\alpha_i\}_{i<k}$. It turns out that not only is there no explosion in the size of Σ but in almost all cases $k \leq 64$. Due to the low value of k, each minterm α_i in $\Psi_{\mathbb{I}}$ is represented internally by the number 2^i and conjunction of predicates is bit-wise-and, complement is bit-wise-complement of numbers, where the type of those numbers is typically `UInt64`.

Product algebra $\mathbb{I}\times\mathbb{S}$. We lift the algebra $\mathcal{B}$ into a multi-terminal BDD algebra, whose terminal algebra is a $\mathbb{I}$. This lifting gives us the implementation of the Cartesian product $\mathbb{I}\times\mathbb{S}$ of $\mathbb{I}$ with $\mathbb{S}$, and allows us to work seamlessly, and parametrically with input predicate conditions in combination with counter conditions. This allows us to leverage symbolic automata algorithms over the alphabet $\mathbb{I}\times\mathbb{S}$. In particular, the basic implementation of the determinization algorithm of CAs uses several underlying support algorithms of symbolic automata modulo $\mathbb{I}\times\mathbb{S}$. In particular, during the main step of the algorithm, mintermization is localized to powerstates, e.g., some powerstate may have a local minterm such as $(\alpha_1 \vee \alpha_7)\wedge\top_{\mathbb{S}}$ if the different cases do not matter locally. However, the determinization algorithm for symbolic automata can not be used "as is" because it will not be able to ditinguish between counter minterms and will overapproximate target powerstates.

4.7 Conclusion and Future Work

We have presented a framework for efficient pattern matching of regexes with bounded repetition, which includes a derivative construction to compile regexes to counting automata, their subsequent determinisation into novel counting-set automata, and a fast matching algorithm. The resources needed to build the CsAs are independent of repetition bounds. It handles a majority of regexes with bounded repetition found in practice, with a much more stable performance than other matchers.

In the future, we intend to explore the limits of the idea of counting sets to enlarge and clearly delimit the class of regexes and counting automata that can be succinctly determinised while preserving fast matching. We also plan to explore possible usage of CsAs as a replacement of classical automata in other applications where automata are used, for instance, as symbolic representations of state spaces. For this, we intend to develop CsA counterparts of essential automata techniques, such as Boolean operations and minimization/size-reduction techniques. We also wish to elaborate on our method for generating texts for stress-testing matchers on regexes with bounded repetition.

'A password is like a toothbrush. You should not share it with anyone.'

unknown

5

Counting in Regexes Considered Harmful

Several methods and tools have been developed that attempt to determine whether a given regex is vulnerable to a ReDoS and to generate a triggering text (also referred to as *evil text* hereafter). Existing ReDoS analyzers [77, 107, 109, 83, 60] focus on the most common family of matchers: those based on the backtracking algorithm. Until now, the automata-based regex engines went unnoticed.

In this chapter, we present the first systematic large-scale study of vulnerability of automata-based matching focused on bounded repetition. We propose a methodology for generating evil texts based on the lightweight ReDoS generator, outlined in Chapter 4. We target mainly matchers based on online DFA-simulation, but our techniques can also be effective with other kinds of automata-based matchers, such as `Hyperscan` (Section 5.3). Our experiments confirm that our generator is the first one effective in finding evil texts for automata-based matchers.

As an example, consider the regex '`%[^\x0d\x0a]{1000}`' (from the database of regexes of the intrusion detection system SNORT [63]), which tells the matcher that when it sees '%', it can accept after seeing exactly 1,000 non-space characters. It is a variation of the regex '`.*a.{`k`}`' from Example 1.3.1 used in practice. The NFA of the regex is heavily non-deterministic and has more than 1,000 states. The minimal DFA has more than 2^{1000} states (it needs to always "remember" all positions of the character '%' within the last seen 1,000 characters). A text on which the DFA would reach many of the DFA states is highly problematic for most matchers, backtracking, as well as online DFA-based. Such a text is, however, also highly specific and the probability of generating it randomly is low (the text must contain sub-strings of the length 1,000 with many different placements of '%'). Our generator is the only automated tool we know that can discover such a text.

Our generator is based on heuristics that generate expensive runs of the DFA of the regex. Besides a general algorithm applicable to any regex, it features a heuristic specialising on bounded repetition, based on an analysis of the counting-set automata (presented in Chapter 4). Especially with extended regexes such as the regex '`%[^\x0d\x0a]{1000}`' from above, it is capable of forcing creation of many large DFA states—the number of these states may be exponential and their size may be linear in the repetition bound, dramatically increasing the matching time.[1]

[1] Bounded repetition may be expressed without the counting, by simply repeating the pattern the needed

We evaluated our generator on a comprehensive database of regexes (from software projects at GitHub [27], network intrusion detection systems [63, 80, 95], detection of security breaches [97], academic papers [110, 102], posts on Stack Overflow [71], and the RegExLib database [78]). We used a set of major industrial regex matchers (`RE2`, `grep`, `Hyperscan` [42, 32, 20], standard library matchers of `.NET`, `Python`, `Perl`, `PHP`, `Java`, `JavaScript`, `Rust`, and `Ruby`), and our matcher `GadgetCA`. We compared the performance of our generator against existing ReDoS generators (`RXXR2` [77], `RegexStatic` [107], `RegexCheck` [109], `Rescue` [83], and `Revealer` [60]). The results of the evaluation substantiate the following conclusions, which are also the main contributions of this chapter:

1. Bounded repetition is an Achilles heel of automata-based matchers and our novel generator is the only one that can effectively generate ReDoS texts for them.

2. On the other hand, without bounded repetition, ReDoS generators have none or negligible success with automata-based matchers.

3. Our new ReDoS generator can indeed generate attacks on practical applications where the performance of regex matching is critical, namely on Snort 3 with enabled `Hyperscan` [63], as well as hardware accelerated regex matching on the NVIDIA BlueField-2 DPU [69]. For both technologies, we achieved a slowdown of regex matching engines by a few orders of magnitude, tested on regexes from real-world Snort rulesets.

Outline. Section 5.1 contains state of the art on ReDoS detection. In Section 5.2, we present our main technical contribution, the ReDoS generator targeting automata-based matchers. This chapter also analyses weaknesses of online DFA-simulation and gives grounds to develop our novel ReDoS generator based on analysing the regex's DFA. Section 5.3 details the experiments, giving evidence of vulnerability of automata-based matching against bounded repetition, including concrete practical implications, and Section 5.4 suggests possibilities of mitigating the implied security risks. Section 5.5 concludes the results.

5.1 Related Work on ReDoS Detection

ReDoS is a form of attack based on a super-linear evaluation of matching regex against a malign text. Several methods and tools have been developed that attempt to determine whether a given regex is vulnerable to a ReDoS and to generate a triggering text (also referred to as *evil text* hereafter). ReDoS [72] vulnerabilities have typically been attributed to excessive use of *backtracking*, as discussed in depth in [26, 25]. Backtracking

number of times, leading to the same DFA. This is, however, impractical and almost never used.

The pitfalls of counting show even in the worst case complexity of the DFA and matching algorithms. In contrast to basic regexes, where the DFA is exponential and the matching time is linear to the size of the regex (when matching by automata algorithms such as online DFA simulation), bounded repetition leads to a doubly exponential DFA and singly exponential matching time. This is because the DFA for a bounded repetition is exponential in the repetition bounds (or their multiple in the case of nested bounded repetitions, as in '`((a{10}){10}){10}`'), which is again exponential in the size of their decadic numerals.

is commonly used in matching engines [91] in order to increase expressiveness at the expense of performance predictability. Backtracking regex matching engines are essentially Turing complete (cf. [62]) and therefore most analysis questions about them are difficult or undecidable. All prior research on ReDoS generators has focused on methods that attempt to generate inputs that essentially cause excessive backtracking at runtime, effectively causing non-termination of matching. Here we summarize main such approaches.

The generator can be either static, dynamic or combination of both based on whether actual regex matching is conducted. Static ReDoS generators are primarily based on the NFA representation of regexes [57] and exploit different techniques, such as *pumping analysis* [57, 76], *transducer analysis* [94], *adversarial automata* construction [109], and *NFA ambiguity analysis* [108]. Such techniques can be sound and even complete for certain classes of regexes. Their main disadvantages are high rate of false positives and ineffectiveness against non-backtracking regex matching engines.

While, dynamic ReDoS generators conduct actual regex matching and use the profiling results to improve next iteration of text generating, e.g., fuzzing [41]. However, the main drawback of the dynamic generators is that they are much more time and space consuming so they may miss ReDoS vulnerabilities in case of complex regexes due to the time or space limits.

`RegexStatic`. The first representative of the static generators is `RegexStatic` [108]. It supports also nonregular features like back-references. It transforms the given regex into a *prioritized-NFA* (pNFA) with prioritized transition function. The prioritized function adds priorities on ϵ-transitions to indicate the order in which a backtracking matcher will traverse them. Then it performs a kind of subset construction when it keeps in a given state track of the states that are reachable with higher priority paths on the same input. The key insight is that a backtracking matcher first tries to match input string using higher priority ϵ-transition and when a backtracking matcher fails to match, it backtracks to take the lower priority ϵ-transitions. Thus, some states are not reachable and make the analysis imprecise and can be removed.

To determine the worst-case matching time, it takes the pNFA and constructs a backtracking tree that simulates the matching process of a backtracking engine. The growth in size of these backtracking trees, in relation to the length of the input string, models the rate of growth of the matching time. The worst-case simulation cost for a regex on an input is then considered as linear, polynomial, or exponential based on how the depth-first search tree is predicted to evolve during backtracking.

It also constructs an *attack automaton* to recognise the language of all exploit strings for a regex causing a backtracking matcher to run super-linear in the length of the text. For example for a regex $E1(E2|E3)^*E4$ automata A_p, A_c and A_s are created such that $L(A_p) = L(E1)$, $L(A_c) = L(E2) \cap L(E3)$ and $L(A_s) = \overline{L((E2|E3)^*E4)}$, where $\overline{L(E)}$ is used to denote the complement of the language $L(E)$. The resulting attack automata A_a is formed by using A_p, A_c and A_s, such that $L(A_a) = L(A_p) \cdot (L(A_c))^+ \cdot L(A_s)$ [106].

`RegexCheck`. The next tool based on the static analysis is `RegexCheck` [109]. It has limited support for extended (nonregular) features. Like `RegexStatic`, it identifies if a regex has linear, super-linear, or exponential time complexity based on its NFA. The NFA is vulnerable if there exists a string such that the number of distinct matching paths from the initial state to a rejecting state is exponential in the length of the string.

It also constructs an attack automaton capturing all those strings that trigger the worst-case behaviour. An automaton $A^E = A_p \cdot (A_1 \cap A_2) \cdot \overline{A_s}$. and the evil string of the automata A is then of the form $s_1 \cdot s^k \cdot s_2$ such that $s_1 \in L(A_1)$, $s \in L(A_1 \cap A_2)$ and $s_2 \in L(A_2)$. It combines static and dynamic analysis to avoid false positives when for each regex is determined a lower bound on the length k where the shortest string s exceeds the time limit t.

RXXR2. RXXR2 [76, 77] constructs an NFA from a given regex and then it searches for instances of a pattern in the NFA using an efficient pattern matching algorithm. Like the tools mentioned above, it is based on the fact that a regex is vulnerable for a backtracking matcher if there exists more than one path through a corresponding NFA for its subexpression. It searches all sub-expressions for exponential vulnerability in a form of $e_1 e_2^* e_3$ where e_1 is a prefix expression, e_3 is a suffix expression, and e_2^* is a vulnerable expression. The result is an attack string $xy^n z$ such that $x \in L(e_1)$, $y \in L(e_3)$ and $xy^n z \notin L(e_1 e_2^* e_3)$.

Since not all states of NFA can be reached during the backtracking matching (a match can be found sooner than all states are discovered) it performs also extended analysis to determine which states "may" and which "must" be reached to avoid false positives. RXXR2, similarly to RegexCheck, is unable to parse extended regexes (except from backreferences, non-capturing groups, greedy quantifiers and lazy quantifiers).

SlowFuzz. A representative of the dynamic fuzzing tools is SlowFuzz [74]. It is based on an evolutionary fuzzer [61] that searches for those inputs that can trigger a large number of edges in the control flow graph of the program under testing. However, it lacks knowledge of regex structures, which may lead to false negatives. The results in [74] compare matching slowdown among individual iterations of the algorithm. Out of the tools mentioned here, it is the most general tool for generating evil texts, since it can handle most of the extended features supported in regexes.

We note that we do not include SlowFuzz into the evaluation since we were not able to run it in our test environment. According to [83], Rescue, which we do include, is more effective than SlowFuzz.

Rescue. One of the tools that combine dynamic and static techniques is Rescue [83]. It uses a genetic search algorithm as a guide. The aim is to find an input string that maximizes the number of matching steps using regex search profiling data. Like SlowFuzz, Rescue can handle also extended grammar (except from set operations and conditionals).

Rescue is based on a three-phase gray-box analytical technique which finds for a given regex a timeout-triggering input string:

- In the *seeding phase*, given a regex and corresponding NFA a genetic (seeding) algorithm searches for a diverse set of strings that cover as many NFA states as possible regardless of the search time. For each string, an *effective prefix* and redundant suffix is kept to guide the cross-over operation and mutation in the *incubating phase*. The effective prefix is a prefix of the string s with a maximal number of characters in s that have been matched and the redundant suffix is the rest of the string s.

- In the *incubating phase*, a genetic algorithm searches for candidates with a maximal ratio between the matching steps and their length. The new candidates are created applying mutation and crossing over operations to the seed strings from the seeding phase.
- In the *pumping phase*, the best candidate with the highest cost-effective ratio are searched. First, all ReDoS-irrelevant characters are removed from the selected string. Then, the algorithm searches for a substring such that its pumping increases the number of matching steps the most. Finally, the best substring is pumped as many times as possible to create a string of the given length l.

For each regex, the maximum length is set to $l = 128$. An output ReDoS string successfully exploiting the ReDoS vulnerability if it takes more than 10^8 matching steps.

`Revealer`. Another tool that combines static and dynamic techniques is `Revealer` [60]. It is inspired by [109]. It converts a regex into an NFA to locate vulnerable subexpressions. Then it dynamically exploits the potential vulnerability by generating attack strings. Unlike fuzzing, it does not mutate input seeds but simulates the matching process of a regex on top of a simplified representation of NFA (*E-Tree*). It tries to generate a common match string for several different paths through the E-Tree. The maximal length of the attack string is limited to 128 characters. Finally, it validates if the attack string takes super-linear matching steps to exclude any false positives.

Finally, let us note that existing generators sometimes aim at extremely severe vulnerabilities, for instance, where a backtracking-based matcher gets completely stuck on a text hundreds of characters long (e.g. [83]). Automata-based matchers do not exhibit vulnerabilities this severe, but they can still be slowed down by orders of magnitude, for which they need a long-enough input text (in the order of megabytes). These are the vulnerabilities that we target.

5.2 ReDoS Generation

We now discuss our ReDoS generator, i.e., a tool that generates an evil text for a given regex. We target primarily automata-based matchers, mainly those based on online DFA-simulation (although, as we show in Section 5.3, our technique works for backtracking matchers as well, and it can be tweaked to cause significant troubles also to `Hyperscan`, which uses NFA-simulation). The generator, combined with a technique that exploits counting presented subsequently in Section 80, is the main technical contribution of this chapter.

Hypothetical matcher. We first discuss a hypothetical matcher, which will serve as a model target for our ReDoS generator described later. The model was created by studying the implementations of the regex matchers in `grep`, `Rust`, `SRM`, and `RE2`. It uses online DFA-simulation with a specific management of the DFA cache, similarly to the mentioned matchers. It uses online DFA-simulation with a specific management of the DFA cache. The matcher does not take into account specific advanced optimizations and implementation techniques used in real performance-oriented matchers. Taking them into account might, of course, improve the performance of the generator for a specific matcher, but our goal is a ReDoS generator that is universal and simple, and

therefore we use a model that captures only the most important common aspects. Despite that, the matchers `grep`, `RE2`, and the standard matcher in `Rust` are quite close to this hypothetical matcher (while `Hyperscan`, which uses the most radical innovations and only NFA-simulation instead of online DFA simulation, is related to it more loosely).

The matching algorithm and its complexity. The hypothetical matcher implements the online DFA simulation with the following management of the cache:

(i) When the cache exceeds some size, it is reset and

(ii) if the cache utilization is too low or is reset too often, the matcher disables the cache completely and reverts to pure NFA-simulation.

Algorithm 3 describes the hypothetical matcher in pseudocode. It simulates a run of the DFA obtained by subset construction from the input NFA $A = (Q, \delta, q_0, F)$ along the input word w. In order to do this without constructing the entire DFA up-front, it uses the class DFA, which constructs DFA transitions and encountered DFA states lazily, on demand, and saves them for further use. Namely, it stores integer IDs of the encountered DFA states (subsets of Q) in a hash table $state2id$, paired with the inverse mapping $id2state$ of the DFA states back to their IDs. A discovered DFA state is identified with the number of the so far identified states plus one (Line 40). The ID of the target state of each used DFA transition is saved in the map $successor$, accessible under the ID of the source state and the symbol on the transition. The map $final$ records whether an ID belongs to a final state.

The i-th character $w[i]$ of the input line is processed in a single iteration of the **for** loop on Line 26. The cost of the iteration depends on whether the DFA transition is in the cache or not. If yes, then $successor[q, w[i]]$ on Line 45 simply returns the ID q' of the successor of the current state ID q. The lookup has a small constant cost (accessing the index $w[i]$ of an array of successors associated with q).

On the other hand, if the DFA transition is not cached, then it must be constructed, which is expensive: The construction requires to iterate through all $w[i]$-transitions originating from the NFA states in the current DFA state S (Line 48). The cost of this iteration depends on the size of S and the number of the used NFA transitions, both of which can be bounded by $|A|$ (the size of A, $|A| = |Q| + |\delta|$). Furthermore, the book-keeping costs of the cache of DFA states, paid after every cache miss on Line 45, is also significant (although dominated by the cost of constructing the transition on Line 48). Looking up a DFA state on Line 37 and adding a DFA state on Line 49 both take time proportional to the size of the DFA state.

The complexity of matching with a high utilization of the cache is therefore approaching $O(|w|)$, but in the worst case, with a low cache utilisation, it increases to $O(|w| \cdot |A|)$. The multiplicative factor $|A|$ may be especially high with extended regexes with the bounded repetition operator, where the size of $|A|$ is linearly dependent on the repetition bounds (this is exponential in the size of the regex, assuming that the bound is given as a decadic or similar numeral). For instance, the NFA for the regex '`.*a.{k}`' needs $k + 1$ states and the DFA obtained by the subset construction has 2^{k+1} states, each of them a set of up to $k + 1$ states of the NFA.[2]

[2]The '`.*`' in the regex is included for clarity, but note that it is redundant in the absence of anchors.

Algorithm 3: Hypothetical matcher

Input: NFA $A = (Q, \delta, s_0, F)$, word w
Output: $\top$ iff $w \in L(A)$, otherwise $\bot$

```
24 dfa ← new DFA()
25 q ← dfa.init({s_0})
26 for i ← 1 to |w| do                                   // O(|w| · |A|)
27     if dfa.final[q] then return ⊤
28     q' ← dfa.get_successor_id(q,w[i])                 //O(|A|)
29     q ← q'
30     if dfa.big() then q ← dfa.init(dfa.id2state[q])
31     if dfa.ineffective() then disable DFA caching
32 return ⊥
33 class DFA:
34     state2id : 2^Q → ℕ; id2state : ℕ → 2^Q;
35     successor : ℕ × Σ → ℕ; final : ℕ → {⊤, ⊥}
36     method get_state_id(S ⊆ Q):
37         q ← state2id[S]                               // O(|S|)
38         if q = None then
39             q ← state2id.cardinality + 1
40             state2id[S] ← q                           // O(|S|)
41             id2state[q] ← S
42             final[q] ← (S ∩ F ≠ ∅)
43         return q
44     method get_successor_id(q ∈ ℕ, a ∈ Σ):
45         q' ← successor[q, a]                          // O(1)
46         if q' = None then
47             S ← id2state[q]
48             S' ← {s' | s ∈ S, s -{a}↦ s' ∈ δ}          // O(|A|)
49             q' ← get_state_id(S')                     // O(|S'|)
50             successor[q, a] ← q'
51         return q'
52     method init(S ⊆ Q):
53         id2state ← state2id ← successor ← final ← ∅
54         return get_state_id(S)
```

The algorithm manages limited resources available for the cache on Lines 30, 31. The cache is reset on Line 30 if it grows beyond some predefined bound (given by the method dfa.`big()`, whose implementation would be matcher-specific). The size of the cache is computed as the sum of sizes of cached DFA states plus the number of cached transitions, $\sum\{|S| : DFA.state2id[S] \neq \texttt{None}\}+|\{(id, a) : successor[id, a] \neq \texttt{None}\}|$ (note that larger DFA states hence contribute more to the size of the cache). Line 31 may then entirely disable caching if the cache is reset too often or if its utilisation is too low (given by a matcher specific implementation of dfa.`ineffective()`). Disabling the cache means reverting to NFA-simulation in which every step must iterate through all NFA states in the current set and all their transitions with the current letter.

Multi-line mode. The matcher described above works in the *single-line mode*. In the *multi-line mode*, the **for** loop on Line 26 is wrapped in an iteration over all lines and every matched line is reported. Importantly, the DFA cache is *not reset* after processing one line, but is re-used when processing subsequent lines.

ReDoS generation algorithm. As follows from the analysis above, our best shot to stress the hypothetical matcher is to attempt to increase its runtime close to $O(|w| \cdot |A|)$ by rendering the cache ineffective and forcing construction of many large DFA states and expensive transitions. For that, recall that every newly discovered DFA state $S \subseteq Q$ is searched for and inserted into the cache, with a cost linear to its size, and subsequently causes a cache miss and forces the construction of a transition on Line 48, with a cost linear to the number of $w[i]$-transitions starting in S. The size of S also determines the cost of looking up and inserting DFA states to the cache on Lines 37, 49. The cost of creating the DFA transition, that is, the number of the NFA transitions, is usually strongly correlated with the size of the source state S (even though it is not precisely determined by it since it depends on the transition relation).

Our aim therefore is to produce a text that discovers many different large DFA states as fast as possible. In other words, we want to force a DFA run (or sequence of runs in the case of multi-line matching) with a high ratio of the sum of sizes of newly discovered DFA states and the text length. We will call this ratio *evilness* of the text. Highly evil texts cause a low cache hit/miss ratio, the cache also fills in quickly and must be reset frequently, and there is a high chance that the utilisation of the cache drops to the point where it is completely disabled.

ReDoS generator overview. Our ReDoS generator constructs a text w with high evilness as a concatenation of lines $w_1 \cdots w_n$, each w_i constructed along a run ρ_i of the DFA from the initial state. Every run ρ_i first takes a shortest possible path through the already visited part of the DFA to a largest discovered but so-far unvisited *start state*, from where it navigates to new unvisited DFA states through DFA transitions chosen according to some successor selection criterion. The run ρ_i is thus a concatenation $\rho_i^1.\rho_i^2$ of a prefix ρ_i^1 through visited states and a suffix ρ_i^2 through unvisited states. The criterion for navigating the second phase, that is, for selecting unvisited successors while constructing the suffix, is a parameter of the algorithm. The basic strategy, called **Greedy**, simply selects the largest unvisited successor (alternatives will be discussed later). This drives the exploration towards large new states. The run ρ_i then ends when it cannot continue to any unvisited and non-final state.

Avoiding final states has the following rationale. Obviously, continuing a line after reaching a final state would be counterproductive because the matcher has already returned *true*. Avoiding final states altogether additionally means that we generate only non-matching lines, which is motivated by the fact that we ideally want texts that are hard for online DFA-simulation-based as well as backtracking matchers. Non-matching lines are generally harder for backtracking matchers. They cannot terminate early after finding a single accepting NFA run but are forced to explore the entire tree of runs over the input line.

Algorithm 4: DFA-based text generation

Input: An NFA $A = (Q, \delta, s_0, F)$, successor selection criterion **Strategy**
Output: evil text w (concatenation of several lines)

55 $q_0 \leftarrow \mathit{DFA}.\texttt{init}(A)$;
56 $\mathit{unvisited}.\mathit{enqueue}(q_0)$;
57 $\mathit{dist}(q_0) \leftarrow 0$;
58 $\mathit{visited} \leftarrow \emptyset$;
59 $w \leftarrow \epsilon$;
60 **while** $\mathit{unvisited} \neq \emptyset$ **do**
61 $q \leftarrow \mathit{unvisited}.\mathit{dequeue\_nearest\_largest}()$;
62 $\mathit{visited}.\mathit{add}(q)$;
63 $w \leftarrow w \cdot \mathit{prefix}(q)$;
64 **while** $\top$ **do**
65 $\mathit{succ} \leftarrow \emptyset$;
66 **for** $a \in \Sigma$ **do**
67 $p \leftarrow \mathit{DFA}.\texttt{get\_successor\_id}[q, a]$;
68 **if** $\mathit{DFA}.\mathit{final}[p] \vee p \in \mathit{visited}$ **then break**;
69 $\mathit{succ}.\mathit{add}(p, a)$;
70 $\mathit{unvisited}.\mathit{enqueue}(p)$;
71 **if** $\mathit{dist}(q)+1<\mathit{dist}(p) \vee \mathit{dist}(p)=\texttt{None}$ **then**
72 $(\mathit{dist}(p), a(p), \mathit{pre}(p)) \leftarrow (\mathit{dist}(q)+1, a, q)$
73 **if** $\mathit{succ} = \emptyset$ **then break** ;
74 $(q', a) \leftarrow \mathit{succ}.\mathit{choose}(\textbf{Strategy})$;
75 $\mathit{unvisited}.\mathit{remove}(q')$;
76 $\mathit{visited}.\mathit{add}(q')$;
77 $q \leftarrow q'$;
78 $w \leftarrow w \cdot a$;
79 $w \leftarrow w \cdot \texttt{\textbackslash n}$;
80 **return** w;

ReDoS generator in detail. We present the algorithm in detail as Algorithm 4. Since constructing the entire DFA may be infeasible due to its size, the algorithm again uses the implicit DFA that is a part of the hypothetical online DFA matcher in Algorithm 3 and thus constructs only those parts of the DFA used to process the generated text.

Every iteration of the while-loop on Line 60 generates one line of the text, the ith iteration generates w_i by constructing the run ρ_i. The algorithm maintains a set $\mathit{visited}$ of identities of DFA states that were visited by some run ρ_i, and a set $\mathit{unvisited}$ of identities of discovered but yet unvisited states. The while loop terminates when there are no states remaining in $\mathit{unvisited}$. To select the starting state q of a run ρ_i (Line 61) and construct the shortest run to q quickly (by the function prefix on Line 63), the algorithm uses a mechanism analogous to that used by the Dijkstra's algorithm for computing the shortest paths from a given source. Every discovered DFA state $p \in \mathit{visited} \cup \mathit{unvisited}$ remembers the last transition in the shortest discovered run to it, namely, the predecessor state $\mathit{pre}(p)$ on the run and the symbol $a(p)$ on its last transition. It also remembers the length (distance) $\mathit{dist}(p)$ of the shortest run. The values of

$pre(p)$, $dist(p)$, and $a(p)$ are updated whenever a transition to the state p is taken (Lines 71, 72). If the run ending by that transition is shorter than the current shortest run the function $prefix(q)$ can then construct the shortest discovered run to q in the form $q_1 \text{-}\{a_1\}\!\!\rightarrow q_2 \ldots q_{n-1} \text{-}\{a_n\}\!\!\rightarrow q_n$ by taking $q_n = q$, $q_i = pre(q_{i+1})$ and $a_i = a(q_i)$ for all $i : 1 \leq i < n$, and return the word $a_1 \ldots a_n$ read along this run. The starting state identity q is chosen (on Line 61). From those which identify the largest DFA states (the state is obtained as $DFA.id2state[q]$), we take one with the smallest distance $dist(q)$.

The suffix of the run, ρ'_i, is where the text supposed to increase the cost of matching is generated. The algorithm navigates through the unexplored DFA states according to the strategy given by the input parameter **Strategy** until the current state q has unexplored and non-final successors p (Line 73). Namely, the for-loop on Line 66 collects into $succ$ all transitions leading to non-final and not yet visited DFA states from the current state q (as pairs consisting of the target state p and symbol a). The particular transition is selected from there according to the criterion **Strategy** on Line 74.

Exploration strategies. The algorithm is parameterized by the strategy of exploration of the unvisited DFA state space, represented by the successor selection criterion **Strategy**. We will consider three variants.

First, **Random** picks a random successor. This produces mostly random but still 'reasonable' texts where reasonable stands for that matching a line never returns false before the line ends as the generated DFA run never leaves the space of useful DFA states. We use the random strategy as a baseline to confirm that the reasoning behind our other two selection criteria, supposed to generate highly evil texts, works, and indeed produces much more evil texts.

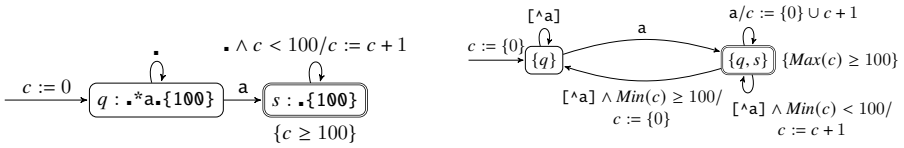
The simpler of the two, **Greedy**, navigates the search towards large DFA states by always choosing the largest successor. The more complex strategy **Counting** is then optimized towards generating texts for regexes with bounded repetition, and it is discussed in the following section.

ReDoS generation for bounded repetition. We will now discuss the specialisation of the ReDoS generator from Section 4 for regexes with bounded repetition. That is, we will specify the successor selection criterion **Counting** used as a parameter **Strategy** of Algorithm 4.

DFAs of regexes with bounded repetition tend to have extremely many extremely large states. This shows even in the worst case complexity of online DFA-simulation (as well as of NFA-simulation), which becomes exponential in the size of the regex (linear in the repetition bounds). The general idea of generating evil texts for bounded repetition is the same as for normal regexes—to force many different and large DFA states. We propose an optimized strategy for navigating towards them.

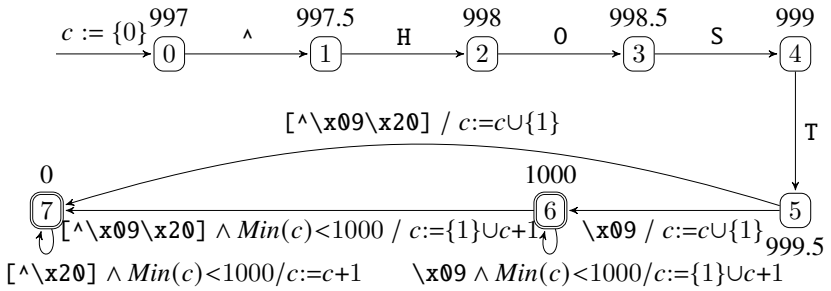
Counting automata. We build the strategy based on the observations of NCA from Chapter 4. Let us recall that a run of an NCA over a word goes through a sequence of configurations, pairs of the form (q, ν) where q is a control state and ν is a counter valuation, a mapping of counters to their integer values.

For instance, one of the NCA runs from Figure 5.1a on the word 'a^{100}' generates configurations $(q, c = 0)$, $(s, c = 0)$, $(s, c = 1)$, $\ldots$, $(s, c = 99)$, but the NCA can postpone



(a) NCA for '`.*a.{100}`'

(b) CsA from determinisation of (a)



(c) CsA with weights for the regex '`^HOST\x09*[^\x20]{1000}`'

Figure 5.1: NCA and CsA. The transitions are labeled by their *guard*, which specifies the input character class ('.' stands for "any character") and possibly restricts counter (or counting set) values, separated by '/' from the counter *update* (an unspecified update means that the value stays the same). In (b) and (c), the notation $\{0\} \cup c + 1$ stands for the set of values obtained by *incrementing* each value in c, *adding* 0, and *removing* values larger than the upper bound of the counter, 100 for (b) and 1000 for (c). The edges denoting initial states are labelled with *initial values* of the counters. Final states are in (a) and (b) labelled with an *acceptance condition* on counters, e.g. $\{c \geq 100\}$ in (a). In (c), the final condition at states 6 and 7 is $Max(c) = 1000$.

the transition into s arbitrarily, leading to different values of c. It is easy to see that one can construct an NFA whose set of states is the set of reachable configurations of an NCA; the runs of such an NFA would go precisely through the same configurations as the runs of the NCA over the same word.

The naive determinisation of the NCA then produces a standard DFA that would be obtained by the subset construction from the induced NFA described above. The states of the DFA are thus sets of the configurations. For the example from Figure 5.1a, a run of the DFA on the word 'a^{100}' would traverse through the following sequence of DFA states (recall that each set of configurations is one state of the DFA):

$$\{(q, c = 0)\},$$
$$\{(q, c = 0), (s, c = 0)\},$$
$$\{(q, c = 0), (s, c = 0), (s, c = 1)\},$$
$$\dots$$
$$\{(q, c = 0), (s, c = 0), (s, c = 1), \dots, (s, c = 99)\}.$$

Our ReDoS generator therefore navigates through a space of such DFA states. The states may be extraordinarily large especially when the NCA configurations within them have many distinct counter values, such as in our example, where the run on the word 'a^{100}' ends in a DFA state where the control state s is paired with 100 values.

Counting-set automata. Our heuristic for navigating through such DFAs towards large states attempts to increase the number of counter values. To do that, we take advantage of our earlier work on determinisation of NCAs into the counting-set automata (CsAs) (discussed in Section 4).

An example of a CsA is the automaton obtained by determinising the NCA from Figure 5.1a, shown in Figure 5.1b. Let us recall that its run on the word 'a^{100}' would generate the following sequence of configurations:

$$
\begin{aligned}
&(\{q\}, c=\{0\}),\\
&(\{q,s\}, c=\{0\}),\\
&(\{q,s\}, c=\{0,1\}),\\
&\dots\\
&(\{q,s\}, c=\{0,\dots,99\}).
\end{aligned}
$$

Note that the sets of values for c precisely correspond to the values of c that s appear with in the run of the DFA shown above. The run-time configurations of a run of a CsA are (encodings of) states of the DFA that would be generated by a run reading the same word.

Navigation towards large counting sets. Since CsA are still small (relative to the DFA), they can be pre-computed and analysed as a whole. We use such an analysis to obtain guiding criteria that lead a run through their configuration space towards configurations with many different counter values. Since runs of CsA simulate runs of DFA, such guiding criteria may be directly used to navigate runs of DFA, as the successor selection criterion **Counting**.

Particularly, in the CsA for the regex, we try to navigate towards cycles that are likely to create large counting sets. For every counter c, every cycle is assigned a weight $weight_c$ that represents an estimate of the maximum counting set for c that iteration of the cycle can generate. The number reflects the following intuitions:

First, since the counting set c can contain only values between 0 and $\boldsymbol{max}_c$, it can have at most $\boldsymbol{max}_c + 1$ elements. Second, the cycle is pumping up the set if 1) it does not reset it, 2) it adds 0 or 1 and also increments the elements of the set (without the increment, it would be only repeatedly adding $0/1$'s to a set already containing it). Third, it is better if only few increments happen in between additions of $0/1$'s. For instance, a cycle that increments the counting set 4 times per every addition of $0/1$ is actually filling it with multiples of 4, hence it can generate a set of the size at most $\frac{\boldsymbol{max}_c+1}{4}$. In summary, the weight of the cycle for the counter c is non-zero only if the cycle does not reset c and increments c at least once, and then it equals $\boldsymbol{max}_c$ multiplied by the number add_cnt_c of additions of $0/1$ to c divided by the number inc_cnt_c of increments of c, i.e.

$$
weight_c = \frac{(\boldsymbol{max}_c + 1) \cdot add\_cnt_c}{incr\_cnt_c}.
$$

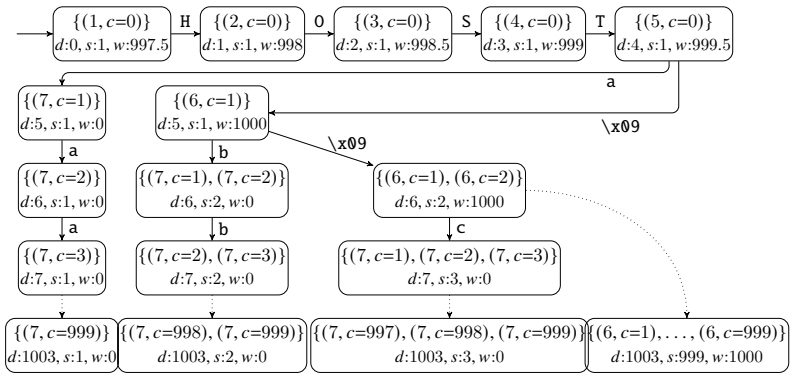


Figure 5.2: DFA states explored by Algorithm 4 on the regex '`^HOST\x09*[^\x20]{1000}`'.

The final weight of a cycle is then computed as a sum of weights for individual counters

$$\sum_{c \in C} weight_c$$

with C being the set of all counters used in the automaton.

The weights of cycles are assigned to states and propagated through the transitions of the CsA. Initially, all states have weight 0. We then process the cycles in the CsA one by one. For each of them, first the weights of all states in the cycle are set to the maximum of their previous weight and the weight of the cycle. The weight of the cycle is then propagated through paths reaching the cycle. Namely, the weight of state r, $weight(r)$, propagates through a transition $q \dashv\{a\}\mapsto r$ so that $weight(q)$ is assigned the maximum of $weight(q)$ and $weight(r) - 0.5$. This is iterated as long as some weight can be increased. In the end, the transitions with heavy target states point in the direction of short paths towards heavy cycles (the *shortness* is achieved through the subtraction of 0.5 for every transition that weight of the cycles is propagated through).

Example 5.2.1. Consider the CsA for the regex '`^HOST\x09*[^\x20]{1000}`' (a simplified regex from SNORT [63]) in Figure 5.1c. States of the CsA have assigned weights according to the algorithm described above. Figure 5.2 shows the tree of DFA states obtained by Algorithm 4.

The underlying NFA would look similar as the CsA in Figure 5.1c, with the difference that there are copies of states 6 and 7 for each value of counter c between 1 and 1000 (and there is a nondeterministic choice over '`\x09`' in states $(6, c{=}i)$ whether to stay in $(6, c{=}i)$ or go to $(6, c{=}i + 1)$).

If traversed using the **GREEDY** strategy (assuming that whenever there is a choice in Figure 5.2 between two DFA states with the same sizes, the strategy picks the left one, e.g., when choosing between $\{(7, c{=}1)\}$ and $\{(6, c{=}2)\}$, **GREEDY** would choose $\{(7, c{=}1)\}$) the traversal would first select the branch that goes to state 7 as soon as possible with DFA states of size 1 (the left-most branch), then it would select the branch with DFA states of size 2 (the second branch from the left), etc., generating the text:

```
HOSTaaa...a\n
HOST\x09bbb...b\n
HOST\x09\x09ccc...c\n
```

```
...
HOST\x09\x09...\x09yy\n
HOST\x09\x09...\x09\x09z\n
HOST\x09\x09...\x09\x09\x09\n
```

The generated text is sub-optimal because it first targets "easy" DFA states of size 1 and explores the most difficult path (with the longest sequence of '`\x09`') only as the last one.

On the other hand, the **COUNTING** strategy avoids this by using the weights computed for the DFA states, which causes that the paths are explored in the reversed order, preferring state 6 because it has a higher weight:

```
HOST\x09\x09...\x09\x09\x09\n
HOST\x09\x09...\x09\x09z\n
HOST\x09\x09...\x09yy\n
...
HOST\x09\x09ccc...c\n
HOST\x09bbb...b\n
HOSTaaa...a\n
```

Indeed, in our experiments, for this regex, `RE2` took 23 times longer to process the text generated by **COUNTING** than the text generated by **GREEDY**. □

5.3 Experimental Results

We have implemented our approach in a C# prototype called `GadgetCA` and evaluated its capability of generating text causing efficiency problem (ReDoS attack) for the state-of-the-art regex matchers especially with regexes that contain a bounded repetition and compared with existing ReDoS generators.

Matchers. We experiment with the matchers introduced in Figure 5.1. We have *automata-based* matchers `grep` [32] (version 3.3), `RE2` [42], `SRM` [82], and the standard regex matcher in `Rust` [31], all four based on online DFA-simulation, `Hyperscan` [20], which uses NFA simulation, and also the prototype matcher `Chipmunk` from Chapter 4, based on counting set automata (Section 80), which specialises in handling bounded quantifiers (`Chipmunk` implements offline CsA-simulation, i.e., it simulates a pre-constructed deterministic CsA on the input text). Then, representing *backtracking* matchers, we have standard library regex matching engines of a wide spectrum of programming languages: `.NET` [65], `Python` [36], `Perl` [96], `PHP` [45], `Java` [30], `JavaScript` [21], and `Ruby` [11]. We note that `grep`, `RE2`, `Rust`, and `Hyperscan` are performance-oriented matchers containing many high- and low-level optimizations.

In Section 5.3, we also experiment with the NIDS SNORT [63], which internally uses `Hyperscan`, and with the hardware-accelerated regex matching engine on the NVIDIA BlueField-2 [69] card.

Except the experiments in Section 5.3, we run our benchmarks on a machine with the Intel(R) Xeon(R) CPU E3-1240 v3@3.40 GHz running Debian GNU/Linux (we run `.NET` tools on the Mono platform [75]).

Size of ReDoS text. In order to avoid low-level noise in the measured times of matchers, we generate texts of the size ~50 MB. We use this value since we observed that at around 50 MB, the ratio between the performance of a matcher on a random text and on a generated ReDoS candidate start to stabilize for many of the used matchers. Larger text sizes may still increase the slowdown, but using them would rise the cost of our experiments beyond what we can manage.

`GadgetCA`. Our generator `GadgetCA` generates a text for a potential ReDoS attack using our approach presented in Section 5.2. In particular, we run the ReDoS text generator for 10 mins or until it completely explores the state space. (We emphasize that generating the ReDoS texts is not a time critical task, since they can be prepared in advance before an attack.) Then, we take the obtained text and copy it as many times as needed in order to obtain a ~50 MB long text.

The particular ReDoS generation algorithm used depends on the chosen search strategy: **Greedy**, **Counting**, **Random**, or **OneLine** (which is yet another strategy used to target Snort's `Hyperscan` in Section 5.3).

Other generators. We compared `GadgetCA` against state-of-the-art generators, which are mainly focused on backtracking matchers (indeed, as far as we know, `GadgetCA` is the first generator targeting nonbacktracking matchers), namely `RXXR2` [77], `RegexStatic` [107], `RegexCheck` [109], `Rescue` [83], and `Revealer` [60].[3] These generators use different algorithms to generate a ReDoS text. The generators may consume excessive time while analysing the regex and generating a ReDoS text, hence, we limited their running time to 10 mins (the same as for our generator). Note that all of these tools are research prototypes, so they do not support all regex features. The generators generate a ReDoS text template in the form of a triple $(prefix, pump, suffix)$ so that a concrete ReDoS text can be obtained by instantiating $prefix \cdot pump^k \cdot suffix$ for some k. Therefore, we set k for each of the ReDoS texts so that $|prefix| + |pump| \cdot k + |suffix| \approx 50\,\text{MB}$.

Dataset. The regexes that we targeted in the experiment were selected from the following sources: (a) the database of over 500,000 real-world regexes coming from an Internet-wide analysis of regexes collected from over 190,000 software projects [27]; (b) the databases of regexes used by *network intrusion detection systems* (NIDSes), in particular, Snort [63], Bro [80], Sagan [95], and the academic papers [110, 102]; (c) the RegExLib database of regexes [78], which is a website dedicated to regexes for various domain-specific languages (DSLs); (d) regexes from posts on Stack Overflow [71]; (e) industrial regexes from Microsoft used for security purposes from Chapter 5; and (f) industrial regexes from TrustPort [97] for detecting security breaches. This gave us a set of 609,992 regexes that we denote as ALL. We then categorized the regexes in ALL into several classes as follows:

SUPPORTED (443,265) is a subset of ALL that contains regexes without features not supported by our tool—e.g., containing look-arounds, back-references, etc.—and regexes

[3] We do not include `SlowFuzz` [74] into the evaluation since we were not able to run it in our test environment. According to [83], `Rescue`, which we do include, is more effective than `SlowFuzz`.

that are not syntactically correct. Moreover, our tool also does not support regexes with the bounded repetition that yield a non-uniform NCA[4] (there were 101 such regexes).

COUNTERS (47,513) is a subset of SUPPORTED containing regexes with bounded repetition. The rest of SUPPORTED is in NOCOUNTERS (395,752).

ABOVE20 (8,099) is a subset of COUNTERS with regexes where the sum of upper bounds of bounded repetition is above 20 (i.e., regexes where the use of bounded repetition may potentially lead to state space explosion). The rest of COUNTERS is put into BELOW20 (39,414).

Methodology. Let us now elaborate on the criteria we use to classify ReDoS attacks. In the literature, we found the following used criteria:

- Shen et al. [83] generate strings of length at most 128 symbols and consider a string a ReDoS if Java's regex library matcher makes at least 10^8 steps on it.
- Davis et al. [26] generate strings of lengths 100 kB–1 MB and call a string a ReDoS if the matcher takes more than 10 s to match it.
- Staicu and Pradel [93] generate pairs of random and crafted strings of an increasing length and measure the differences of the times the matcher takes for the random and the crafted string in each pair, obtaining a sequence $d_1, d_2, \ldots, d_n$. They consider a crafted string a ReDoS if $d_1 < d_2 < \cdots < d_n$.
- Rathnayake and Thielecke [77], Wüstholz et al. [109], and Weideman et al. [108] define that a regex is ReDoS-vulnerable if it meets some condition that causes super-linear behaviour (they do not examine the run time of the matchers in detail).

We base our ReDoS criteria on the criteria in [26], but normalize it w.r.t. the significantly lower average matching times for automata-based matchers ([26] only considers backtracking matchers). Our ReDoS criteria are the following:

- **>10s**: the matching takes over 10 s, (corresponds to the throughput of <5 MB/s),
- **>100s**: the matching takes over 100 s (corresponds to the throughput of <0.5 MB/s),
- **>100×AVG$_{\text{REGEX}}$**: the matching takes at least 100 times more than the time for matching of a random ~50 MB-long text on the given regex (computed as the average of times of 10 different random texts) by the given matcher (the motivation for this is that a user has some idea about the average performance of the matcher on a regex he created and tested), and
- **>100×AVG$_{\text{MATCHER}}$**: the matching takes at least 100 times more than the average time for matching a random ~50 MB-long text on the given matcher across all regexes without the '^' and '$' anchors (regexes with such anchors allow to quickly determine non-match and skip the rest of the text). Average matching times (in seconds) for the matchers are given in Table 5.1.

[4] Due to the technical difficulty of characterizing such regexes and the relatively small number of regexes affected by this, we refer the interested reader to the description in [98, Section 6.4].

Table 5.1: The average matching time [s] of a random 50 MB-long text for each of the matchers (averaged over all regexes).

grep	hyper-scan	re2	srm	ca	rust	ruby	php	perl	python	java	node	.NET
0.04	0.07	0.14	1.02	1.32	0.07	2.13	3.10	0.09	0.69	1.11	0.93	2.59

Table 5.2: Numbers of regexes from ABOVE20 for which various generators successfully generated **>100s**-ReDoS texts. Red (darker) colour emphasizes higher numbers. For each ReDoS criterion, matchers are split into groups based on their types.

Generators		>100s-ReDoS attacks												
		grep	re2	rust	srm	hyper-scan	ca	ruby	php	perl	python	java	java-Script	.NET
GadgetCA	GREEDY	192	72	76	238	0	61	1087	1408	56	200	215	210	390
	COUNTING	216	110	96	272	0	45	1724	1979	89	218	242	211	419
	RANDOM	126	28	48	123	0	46	682	885	60	160	181	111	334
	ONELINE	192	17	32	23	0	56	333	40	187	433	414	378	584
RXXR2		7	0	2	0	0	1	24	0	4	30	11	11	34
RegexCheck		14	0	2	0	0	0	7	1	1	9	8	4	16
RegexStatic		34	1	5	0	0	8	160	63	69	262	253	243	285
Rescue		12	0	3	0	0	2	23	3	4	23	13	12	27
Revealer		16	1	1	5	0	4	38	14	22	56	50	46	71
random text		52	4	11	17	0	82	33	47	23	109	162	36	231

Summary of results. Let us quickly summarize results obtained in our experimental evaluation, described in detail in the following sections:

R1: Regexes with bounded repetition with higher bounds are potentially vulnerable to ReDoS attacks even for automata-based matchers.

R2: If a regex does not contain counting, it mostly cannot be used to perform a ReDoS attack on automata-based matchers.

R3: Our informed exploration strategy **COUNTING** is better at generating ReDoS texts than the (less informed) strategies **GREEDY** and **RANDOM**.

R4: Other state-of-the-art ReDoS generators are not able to generate ReDoS text for automata-based matchers.

R5: Our techniques can be used to attack mature real-world security solutions.

R1: Vulnerability of counting regexes. In our first experiment, we confirmed that the use of bounded repetition with a higher bound in regexes creates a possible attack surface for ReDoS even for online DFA-simulation-based matchers. We used the ABOVE20 set of regexes and tried to generate ReDoS attacks using `GadgetCA` and other matchers using the methodology described above.

First, see Table 5.2, which shows how many successful **>100s**-ReDoS texts different settings of `GadgetCA` were able to generate for online DFA-simulation-based matchers.

Table 5.3: Numbers of regexes from ABOVE20 for which various generators successfully generated **>10s**-ReDoS texts.

Generators		>10s-ReDoS attacks												
		grep	re2	rust	srm	hyper-scan	ca	ruby	php	perl	python	java	java-Script	.NET
GadgetCA	GREEDY	1058	703	274	311	1	135	5050	6580	837	1027	485	955	2629
	COUNTING	1181	1116	295	391	3	121	5440	6289	1294	1503	532	1317	3000
	RANDOM	713	135	259	242	1	106	4405	5389	361	523	385	410	2025
	ONELINE	576	17	78	30	6	130	540	69	402	678	637	485	1448
RXXR2		11	0	2	0	0	1	26	0	5	33	12	13	35
RegexCheck		25	0	3	0	1	0	7	3	7	18	15	9	36
RegexStatic		78	1	9	0	0	19	182	70	78	287	274	254	333
Rescue		11	0	3	0	0	4	24	2	5	26	13	13	28
Revealer		24	1	7	0	0	14	51	19	31	69	61	51	75
random text		153	10	70	27	2	137	175	47	147	272	255	228	698

Notice that we were able to generate 216 ReDoS texts for `grep`, 110 ReDoS texts for `RE2`, 96 ReDoS texts for `Rust`, and 272 ReDoS texts for `SRM` (using the **COUNTING** strategy).

Next, in Table 5.3, you can see data for the weaker ReDoS criterion **>10s**. The number of generated successful ReDoS-texts is significantly higher: 1,181 for `grep`, 1,116 for `RE2`, 295 for `Rust`, and 391 for `SRM` (all using the **COUNTING** strategy).

Under both ReDoS criteria above, the **COUNTING** strategy achieves the best results for online DFA-simulation-based matchers and, moreover, for the **>10s** criterion also for backtracking matchers. Further, note that **GREEDY** also obtains significantly better results than **RANDOM**, proving that our informed search strategies are better in generating hard text than uninformed search, giving the positive answer to **R3**. In the following, we will therefore only consider the **COUNTING** strategy. The table also shows that `Hyperscan`, `SRM`, and `Chipmunk` are more robust towards being attacked by our ReDoS texts: `SRM` has a special support for counters and `Chipmunk` is a matcher that uses counting set automata (Section 80). We will discuss `Hyperscan` in Section 5.3.

In Table 5.4 or Figure 5.4, we provide a comparison of the number of **>100×AVG** MATCHER-ReDoS texts generated by the tools. Again, note that a slowdown of >100 times wrt. the global average for the matcher was achieved on many regexes for online DFA-simulation-based matchers (2,457 for `grep`, 742 for `RE2`, 1,016 for `Rust`, and 300 for `SRM`). Since the global average matching time for `PHP` was 3.1 s and we used the timeout of 300 s for matchers, in this table, the `PHP` column contains the number of timeouts instead. A more detailed analysis for other slowdown ratios is in Figure 5.3. Notice that although `Hyperscan` looks almost invincible in the results in Table 5.2, we are able to slow it down by a factor of 10–50 in many instances (543).

On the other hand, Table 5.5 or Figure 5.5 compare the numbers of generated **>100×AVG**$_{\text{REGEX}}$-ReDoSes. In this case, a slowdown of >100 times wrt. the average time for the matcher and the regex was also achieved often for online DFA-simulation-based matchers (1,157 for `grep`, 1,465 for `RE2`, 1,066 for `Rust`, and 279 for `SRM`).

We conclude that many counting regexes can be successfully attacked using ReDoS texts created by our generator.

Table 5.4: Numbers of regexes with successfully generated **>100×AVG$_{\text{MATCHER}}$** texts.

Generators		>100×AVG$_{\text{MATCHER}}$-ReDoS attacks												
		grep	re2	rust	srm	hyper-scan	ca	ruby	php	perl	python	java	java-Script	.NET
GadgetCA	GREEDY	1744	15	19	96	20	44	261	39	382	368	330	316	433
	COUNTING	2457	742	300	1016	5	67	1355	1596	1473	277	279	258	416
	ONELINE	1796	17	99	23	20	53	322	34	441	448	405	379	521
	RANDOM	2033	120	122	289	3	46	348	388	412	176	177	117	258
RXXR2		13	0	0	2	0	1	24	0	5	30	10	10	34
RegexCheck		104	0	0	5	1	0	7	1	7	11	8	4	14
RegexStatic		93	1	0	9	1	7	159	50	80	263	253	243	279
Rescue		12	0	0	3	0	2	23	2	5	23	13	12	26
Revealer		24	1	0	7	0	3	37	11	32	59	50	46	62

Table 5.5: Numbers of regexes with successfully generated **>100×AVG$_{\text{REGEX}}$**-ReDoS texts.

Generators		>100×AVG$_{\text{REGEX}}$-ReDoS attacks												
		grep	re2	rust	srm	hyper-scan	ca	ruby	php	perl	python	java	java-Script	.NET
GadgetCA	GREEDY	878	14	57	12	23	0	164	9	174	232	190	194	203
	COUNTING	1157	1465	1066	279	2	3	1085	796	1252	407	142	140	171
	RANDOM	1066	320	292	130	0	0	153	156	266	91	63	60	72
	ONELINE	966	15	57	16	23	0	199	9	208	277	232	228	238
RXXR2		1	0	2	0	0	0	10	0	4	22	8	8	20
RegexCheck		4	0	4	0	0	0	3	0	0	4	3	2	2
RegexStatic		47	5	5	0	0	0	80	14	49	137	125	134	90
Rescue		1	2	4	0	0	1	12	2	6	15	7	6	14
Revealer		2	0	2	0	0	0	8	0	8	18	6	19	13

R2: Regexes without counting. The second experiment shows that when targeting automata-based matchers, it is indeed important to exploit counting.

Since the set SUPPORTED is too large for us to run a ReDoS generator for each regex, we use a quick filter based on the intuition that ReDoS in these matchers is caused by generating many large DFA states. Hence we run DFA construction for each regex from the set. If the construction terminates with less than 1,000 states, we consider the regex safe. After 1,000 DFA states, the construction is stopped, and the regex is marked as possibly vulnerable. This test is quick, since constructing 1,000 DFA states is fast, and the vast majority of the regexes have even much smaller DFAs.

To assess the accuracy of the test in predicting that a regex is not vulnerable for automata-based matchers, we apply the test on the regexes from ABOVE20 for which we did manage to generate a ReDoS text for automata-based matchers (cf. the experiment in **R1**). From ~2,000 of them, only `grep` and `Rust` had cases with DFA smaller than 1,000 states, namely 24 cases, 6 for `grep` and 18 for `Rust` (`RE2`, `Chipmunk`, `Hyperscan`, and `SRM` had none). These counterexample cases witness that our filter is not always right, at least for `grep` and `Rust`, and ReDoS with automata-based matchers might be possible even with small DFA. Still, the scarcity of these cases confirms that the test is

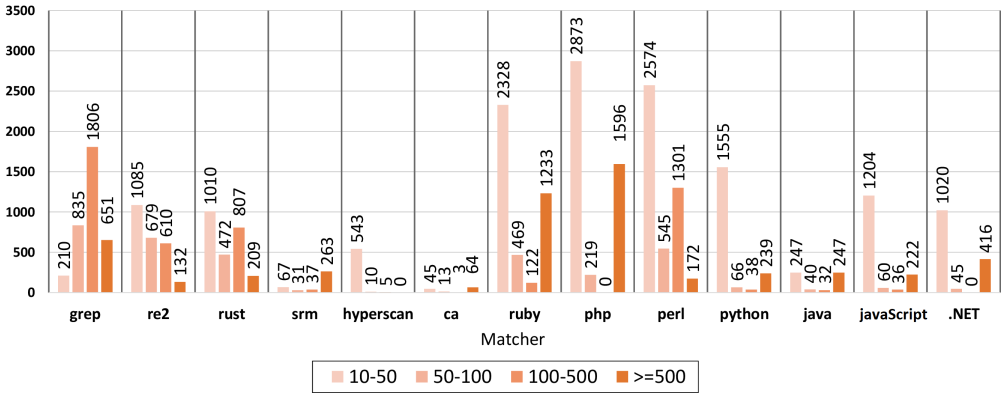


Figure 5.3: Histogram of ratios between times of matchers for random and ReDoS text generated by `GadgetCA`.

a good predictor even for `grep` and `Rust`.[5]

Running the test on SUPPORTED resulted in the following numbers of regexes with DFAs with >1,000 states:

NOCOUNTERS	BELOW20	ABOVE20
175 (0.04 %)	343 (0.8 %)	1,600 (20 %)

We then used `GadgetCA` to generate ReDoS candidates for the regexes in NOCOUNTERS ∪ BELOW20 whose DFAs had more than 1,000 states. Only 7 regexes caused **>100s**-ReDoS for automata-based matchers, two for `grep`— '`\^.{20}\$`' and '`\^_.{19}\$`' (note that both also contain "higher bounds" for the quantifiers)—and 5 for `SRM`. A **>10s**-ReDoS was caused by 24 regexes for `grep` and ~6 regexes for each of `RE2`, `Rust`, and `SRM`. The relative sizes of the sets indicate that regexes without higher repetition bounds are much less vulnerable to ReDoS for automata-based matchers (518 vulnerable from 435,166 in NOCOUNTERS ∪ BELOW20 while 1,600 vulnerable from 8,099 in ABOVE20).

R4: Comparison with other generators. Our next experiment confirms that our generator can create new ReDoS attacks much more effectively than existing tools.

First, compare the middle part of the left- and right-hand side of Figure 5.2. For other generators, the ten-fold stronger **>100s**-ReDoS criterion makes almost no difference: they cannot find and exploit the features of the regex that make the matchers slow down (both for automata-based and backtracking matchers). The same holds for the **>100×AVG**$_{\text{MATCHER}}$ and **>100×AVG**$_{\text{REGEX}}$-ReDoS criteria in Figure 5.4.

Second, compare the bottom part (random text) with the middle part of the table. For counting regexes, a random text is in the majority of cases actually better in creating a ReDoS than current state-of-the-art ReDoS generators (only `RegexStatic` can keep up with the random text on some matchers). Relating this to `GadgetCA` in

[5]The 24 cases are probably caused by specific implementation techniques or different interpretation of the regexes. The 18 cases of `Rust` seem to be related to handling of large character classes (`\w` appears in all 18 cases).

the top part of the table reveals that the numbers of successfully attacked regexes for the two criteria differ significantly, hence `GadgetCA` indeed succeeds in exploiting the critical feature of the regex.

Third, the comparison of the top part with the middle part of the table shows that `GadgetCA` significantly outperforms other matchers on online DFA-simulation-based matchers and most of the other generators even on backtracking matchers (the only exception being `RegexStatic`, which is comparable on some backtracking matchers).

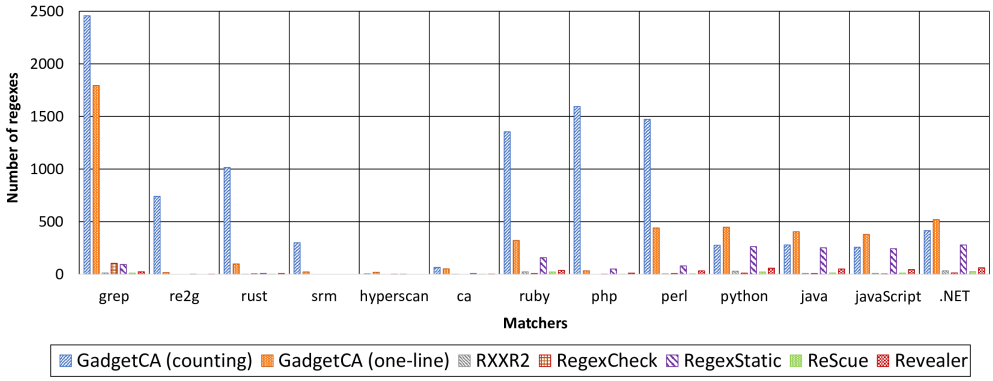


Figure 5.4: Histogram of numbers of regexes with successfully generated **>100×AVG$_{\text{MATCHER}}$**.

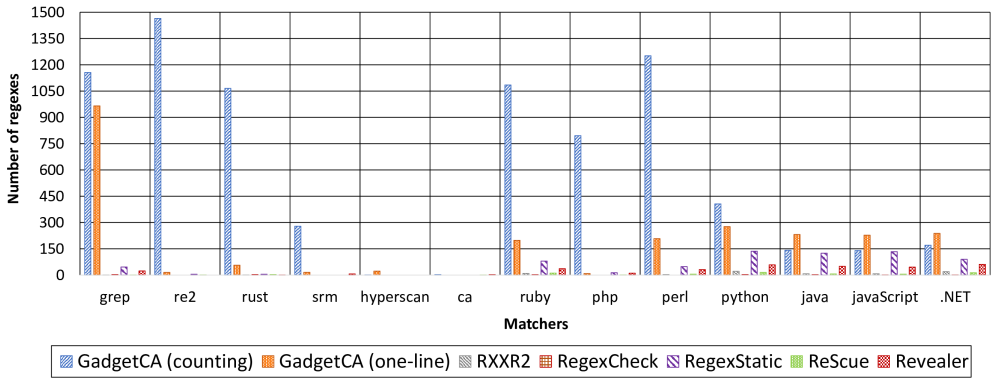


Figure 5.5: Histogram of numbers of regexes with successfully generated **>100×AVG$_{\text{REGEX}}$**.

R5: Real-world security solutions. Our final experiment asks whether the results obtained in **R1** carry over to real-world security solutions, which should be prepared for being targeted by (Re)DoS. We carried out an extensive evaluation of the abilities of SNORT 3 [63], a popular and often used NIDS, which internally uses `Hyperscan`, to withstand ReDoS attacks generated by `GadgetCA`. Instead of using some of the previously introduced datasets, which might contain regexes created by people unaware

of the dangers of ReDoS, we used regexes from rulesets provided with Snort, which are written by security experts and tested in production. In particular, we used regexes from the following four rulesets: (i) Emerging Threats Pro, (ii) Emerging Threats 3CORESec (versions 157 and 164), and (iii) Talos LightSPD (version 2021-03-11-001). We call the obtained set of 1,112 PCRE regexes SNORT (from the original 22,425 original regexes we removed 16,094 regexes not supported by our tool, and then filtered the 1,112 regexes with quantifier bounds at least 20). The experiment was run in two different settings: (i) on a commodity `x86_64` machine with Snort using `Hyperscan` and (ii) on a computer with an NVIDIA BlueField-2 card [70], which provides its own hardware-accelerated regex matching solution.

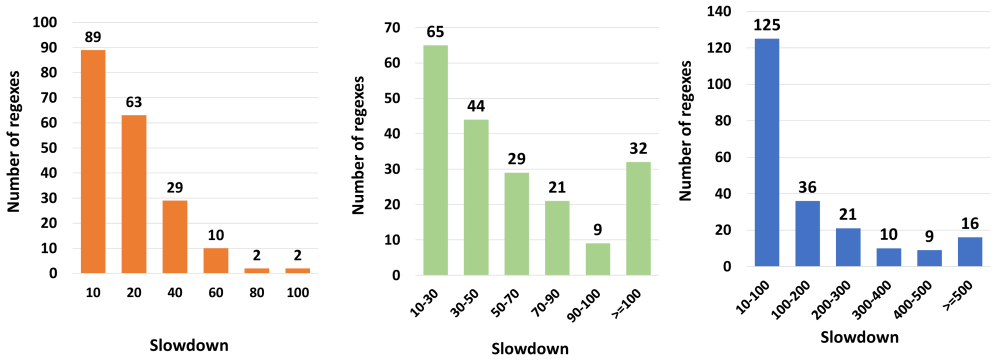
Modified ReDoS generator. In this experiment, we use a modified version of our ReDoS generator for the reason that although `Hyperscan`, used within Snort, can be counted as an automata-based matcher, it is not based on online DFA-simulation. Experiments discussed in the previous sections indeed show that our ReDoS generator, which targets mainly online DFA-simulation, is only mildly successful with `Hyperscan`. We therefore use here a modification of `GadgetCA` tailored for `Hyperscan`.

We specifically target the following coarse abstraction of `Hyperscan`'s matching algorithm: the regex is split into a sequence of *sub-strings* (not containing any regex operator) and *sub-regexes* (or a choice of such sequences) so that a word is matched if it is a concatenation $w = v_1 \cdots v_n$ of the sub-strings of the given regex and words matched by the sub-regexes. The first phase of matching tests whether w contains all the sub-strings in the right order, by an extension of the Boyer-Moore algorithm. The second phase tests whether the remaining sub-words are matched by the respective sub-regexes. The opportunity for slowing `Hyperscan` down is in the second phase, which uses NFA-simulation to match the sub-expressions.[6]

We therefore aim at generating evil texts that contain the needed sequence of sub-strings and therefore pass the first phase of matching, and where the second phase is also hard. To do that, we use our generator to get a single evil word u over a run that takes the CsA from the initial to a final state. The word is essentially generated by the first iteration of the while-loop on Line 64 of Algorithm 4 parameterised with the strategy **Counting** (a single CsA run that aims at maximising the sizes of counting sets). The word u is then iterated to get the output text $w = uuuu \ldots$ of the required length.

A word w generated this way is likely to be evil for the following two reasons: (i) every occurrence of u in w contains all sub-strings, generating many possible splits of w into the sub-strings and the parts to be matched by the sub-regexes; (ii) the word u, generated by our generator, is likely to force large DFA states, expensive for NFA simulation. Note also that, unlike for online DFA simulation, it does not matter that the encountered DFA states are likely to be found repeatedly in the repeating instances of u since NFA simulation is not caching the DFA.

[6] Our abstraction of `Hyperscan` is obviously coarse, but simple and sufficient for our needs: to show that methods similar to those for online DFA-simulation can be used to find vulnerabilities of `Hyperscan` too. A specialised ReDoS generator based on a more thorough analysis of `Hyperscan`'s algorithm might yield better results, but is already out of the scope of this thesis.



(a) Snort 3@Hyperscan (1,500 B)

(b) Snort 3@Hyperscan (9,000 B)

(c) NVIDIA BlueField-2

Figure 5.6: Histograms of slowdowns for Snort 3 with `Hyperscan` (packet sizes 1,500 B and 9,000 B) and BlueField-2 regex matching for ReDoS texts over random texts.

Snort with `Hyperscan` on `x86_64`. We installed Snort 3 with enabled performance monitor and `Hyperscan` on a commodity `x86_64` machine (we used Intel i7-10510U CPU@1.80 GHz with 4 Hyper-Threading cores). Then, we were running Snort on 100 MB-large `PCAP` files with random and ReDoS IPv4 traffic that we generated and captured the processing time of the regex matching engine, as provided in the output of the performance monitor module. We ran two experiments with two different sizes of IP packets for two selected Ethernet frames' MTUs: 1,500 B and 9,000 B (we note that bigger sizes of the payload could be used to attack Snort with TCP reassembly turned on). See Figures 5.6a and 5.6b for the slowdown that we achieved with our ReDoS text over random text.

The histograms clearly show the Snort rulesets we used contain many possibilities for slowing Snort down (see Section 5.3 for the most vulnerable regexes). In particular, using packet size of 1,500 B, in 43 cases we achieved a slowdown of over 40×, with 2 regexes slowing the matcher down over 100×. The number of vulnerable regexes is even higher for the packet size 9,000 B: 91 regexes yield a slowdown of over 50× and 32 regexes over 100×.

We contacted the development team of Snort and did the responsible disclosure of the discovered vulnerable regexes. Snort development team stated that the vulnerability is stemming from the `Hyperscan` library, and they mitigate it by restricting the length of packets on which the matching is performed as well as by using timeouts (the standard configuration of Snort comes with the backtracking-based PCRE engine enabled, which is, however, even more prone to attacks). This might, however, lead to skipping the malicious content that can be presented at the end of the packet/data, making the NIDS ineffective: malicious packets may get passed to applications behind the NIDS.

NVIDIA BlueField-2. In the second part of this experiment, we used an NVIDIA BlueField-2 data processing unit (BF2) MBF2H332A-AEEOT [69], which integrates eight 64-bit ARMv8 Cortex-A72 cores and houses two 25 GbE interfaces. BF2 provides hardware-accelerated regex matching capabilities, accessible via NVIDIA's *data plane*

development kit (DPDK) [70]: in our experiments, we used the regex compiler `rxpc` and the testbed for the regex matching engine called `rxpbench`. In this experiment, we ran `rxpbench` on blocks of random and ReDoS texts of the length 100 GB (this time, we did not need to chunk the texts into packets and provided the text directly in memory) and measured the throughput of the matcher. We measured that the regex matching engine itself enables in-memory processing at ~40 Gbps. For the evaluation, we used a subset of SNORT rules containing 617 regexes that we name SNORT-BF2 (we took all regexes from SNORT that could be compiled by `rxpc`, which does not support some advanced features of PCRE, such as negative look-ahead).

See Figure 5.6c for histograms of slowdowns we obtained with our ReDoS text as compared to random text. Observe that we obtained a slowdown of more than 100× on the ReDoS text in over 92 cases. Moreover, for 16 cases, we obtained a slowdown over 500× (with the highest slowdown ratio being 2,194×). See Section 5.3 for a list of regexes on which we obtained the largest slowdown. We have reported the vulnerability to NVIDIA, which confirmed it to be caused by a conceptual limitation of their regex matching engine. We plan to cooperate on a possible mitigation.

Our results indicate that ReDoS attacks are in general successful in slowing down the throughput of the most recent hardware utilized for NIDS in the industry. Moreover, we emphasize that for a successful ReDoS attack on an NIDS, it suffices to have a single vulnerable rule in the used rulesets.

Attacks on real-world security solutions. In Tables 5.6 and 5.7, we provide examples of regexes for which we managed to obtain a significant slowdown of SNORT (with `Hyperscan` as the regex matching engine) and the NVIDIA BlueField-2 DPU respectively.

5.4 Mitigation Techniques

Standard techniques for mitigation of ReDoS attacks are the following: (i) setting a resource limit (e.g., a timeout) and (ii) limiting the size of the input (e.g., to the first 100 characters) of the regex matcher. Although such techniques can avert the scenario of a server becoming unresponsive, they leave a part of the input traffic not classified and potentially harmful or unnecessarily dropped. A mitigation specific for regexes with the counting operator is to substitute it by the star '`*`' operator, which over-approximates the language of the original regex (this might yield other issues, such as increasing the number of false positives in an NIDS).

There are, however, two ways how users of regex matchers can mitigate the attacks without the mentioned disadvantages:

1. Use our ReDoS generator `GadgetCA` to evaluate whether a regex is ReDoS-vulnerable.
2. Use a matching algorithm that can handle counting efficiently, the one implemented in the tool CA or possibly also `SRM` (these matchers are still too immature to be used in production, but an efficient implementation of the techniques they use within `RE2` or `Hyperscan` should give rise to a robust regex matching solution).

5.5 Conclusion

We have shown that nonbacktracking automata-based regex matchers, which are sometimes suggested as a mitigation of ReDoS, are still ReDoS-vulnerable. We have developed a method for constructing inputs for these matchers that make them perform poorly and cause significant slowdown on a large class of regexes, in particular those with counting.

In future, we plan to focus on developing robust regex matchers that could prevent these kinds of attacks. A first proof of concept is the matcher CA from Chapter 5, but the class of counting regexes it support is quite restricted; we will therefore explore formal models that can deal with more general classes of counting regexes efficiently.

Table 5.6: Slowdown of regex matching in Snort3 with Hyperscan on x86_64.

SID	Slowdown (MTU=9000B)	Slowdown (MTU=1500B)	Regex
46310	213.95	78.89	[?&]u=[^&\s]{35}
31068	172.32	50.49	<hostname>.{0,250}[\x60\x3b\x7c\x24\x28\x26]
2644	165.81	65.57	\(\s*TIMESTAMP\s*(\s*(\x27[^\x27]+'|\x22[^\x22]+\x22)\s*,)\s*((\x27[^\x27]{1000,})|(\x22[^\x22]{1000,}))
13364	163.52	71.15	src\s*\x3D(3D)?\s*['"][^'"]{244}
19925	160.95	58.7	value\s*=\s*[\x27\x22][^\x27\x22]{257}
2102614	157.95	52.68	TIME_ZONE\s*=\s*((\x27[^\x27]{1000,})|(\x22[^\x22]{1000,}))
17659	157.41	79.18	\s*\x28(\x27[^\x27]{64}|\x27[^\x27]*\x27\s*,\s*\x27[^\x27]{64})
2611	157.39	49.67	USING\s*((\x27[^\x27]{1000})|(\x22[^\x22]{1000}))
46309	152.34	65.7	[?&]p=[^&\s]{260}
39982	145.5	55.61	[?&]sn=[^&]{129}
2651	140.95	51.26	NUMTO(DS|YM)INTERVAL\s*\(\s*\d+\s*,\s*((\x27[^\x27]{1000,})|(\x22[^\x22]{1000,}))
2102699	138.15	49.25	TO_CHAR\s*\(\s*SYSTIMESTAMP\s*,\s*(\x27[^\x27]{256}|\x22[^\x22]{256})((
19121	136.82	63.89	SET\s*EXPLAIN\s*FILE\s*TO\s*[\x22\x27][^\x22\x27]{927}
2640	135.24	56.41	\(\s*(\x27[^\x27]*'|\x22[^\x22]+\x22)\s*,\s*(true|false)\s*,\s*((\x27[^\x27]{1000,})|(\x22[^\x22]{1000,}))
15114	135.06	51.46	embed src=\s*(\x27[^\x27]{1000}|\x22[^\x22]{1000}|[^\s\x22\x27]{1000})
16516	121.4	44.42	sys\x2eolapimpl\x5ft\x2eodcitablestart\x28[^\x2c]+\x2c[^\x2c]+\x2c\s* \x27?[^\x2c\x27]{303}
29184	120.92	54.3	encoding\x3D[\x22\x27][^\x22\x27]{1024}
14991	120.65	61.81	select\s+xmlquery\s*\x28\s*(\x27|\x22)[^\x27\x22]{512}
43005	120.52	35.49	[?&]psk=[^&]{256}
29185	118.76	50.13	version\x3D[\x22\x27][^\x22\x27]{1024}
33310	117.95	54.87	\x3C\x21ENTITY\s+.*\s+\x22\x26[^\x22] {700}
27808	110.1	29.94	\x2f\?[a-f0-9]{60,66}
42078	108.17	43.24	[?&](cmd|pwd|usr)=[^&]{64}
2488	106.01	43.94	name=\s*[^\r\n\x3b\s\x2c]{300}

Table 5.7: Slowdown of regex matching at an NVIDIA BlueField-2 card.

SID	Thourghput on Random Text [Gbs]	Thourghput on Redos Text [Gbs]	Slowdown	Regex
2046	41.24	0.02	2,193.76	/\sPARTIAL.*BODY\.PEEK\[[^\]]\1024\/
19213	41.19	0.02	1,681.04	/Subject\x3a\x20[^\n]*\x3fQ\x3f[^\n]{512}/
17367	40.30	0.03	1,174.83	/\d{3}\s+[^\n]{1019}/
6507	41.09	0.04	957.74	/\x2fnds[^\r\n]{1000}/
1021	41.21	0.04	956.06	/\s{230,}\.htr/
20241	40.66	0.04	947.72	/Oid\x3D[^\x0D\x0A]{1000}/
15489	40.58	0.04	920.28	/\x3cimg[^\x3e]*src\x3d(\x22|\x27)?[^\x22\x27\s]{300}/
3547	40.79	0.05	829.08	/php.*\x3f[^\n]{256}/
25586	41.03	0.06	732.67	/host=[^&]{1024}/
8060	41.31	0.06	728.49	/GET\s\x2f[^\r\n]{900}/
31354	41.14	0.06	656.15	/\x28\x3f\x3d[^)]{300}/
3149	41.22	0.06	655.34	/object\s[^>]*type\s*=\s*[\x22\x27][^\x22\x27]*\x2f{32}/
17568	41.11	0.06	641.29	/\w{3}\x25\x30\x30[^\r\n]{2000}/
4127	41.15	0.08	545.82	/\x2fnds\x2f[^&\r\n\x3b]{500}/
38287	40.97	0.08	543.40	/akey=[^&]{500}/
18484	41.14	0.08	536.42	/https?\x3a\x2f\x2f[^\n\r]{1000}/
43545	41.22	0.08	485.54	/-group[^\r\n\s]{1280}/
33310	40.96	0.09	469.76	/\x3C\x21ENTITY\s+.*\s+\x22\x26[^\x22]{700}/
2701	41.20	0.09	434.18	/sid=[^&\x3b\r\n]{255}/
2107	41.21	0.10	427.53	/\sCREATE\s[^\n]{1024}/
18579	41.18	0.10	426.76	/(Context|Action)\x3D[^\x26\x3b]{1024}/
20889	40.87	0.10	419.58	/<\s*valitem[^>]*\s(value|name)\s*=\s*([\x22\x27])[^\x22\x27]{104}/
2826	41.28	0.10	416.17	/(\(\s*(\x27[^\x27]*\x27|\x22[^\x22]+\x22)\s*,\s*(\x27[^\x27]{1075,}|\x22[^\x22]{1075,})|\(\s*(\x27[^\x27]{1075,}|\x22[^\x22]{1075,})|(\(\s*((\x27[^\x27]*\x27|\x22[^\x22]+\x22)\s*,\s*){2}(\x27[^\x27]{1075,}|\x22[^\x22]{1075,}))/
2826	40.73	0.10	410.57	/(\(\s*(\x27[^\x27]*\x27|\x22[^\x22]+\x22)\s*,\s*(\x27[^\x27]{1075,}|\x22[^\x22]{1075,})|\(\s*(\x27[^\x27]{1075,}|\x22[^\x22]{1075,})|(\(\s*((\x27[^\x27]*\x27|\x22[^\x22]+\x22)\s*,\s*){2}(\x27[^\x27]{1075,}|\x22[^\x22]{1075,}))/
21671	41.16	0.10	403.94	/zip\x3a\x2f\x2f[^\x0A\x20\x09\x0B\x0C\x85\x3E\x3C]{400}/
20240	41.20	0.11	375.87	/Template\x3D[^\x0D\x0A]{1000}/
27940	41.08	0.11	374.44	/password=[^\x26]{1024}/
2103070	41.00	0.11	361.25	/\sFETCH\s[^\n]{500}/
36195	41.21	0.12	338.07	/actserver=[^&]{982}/
36196	40.86	0.12	335.47	/actserver=[^&]{987}/

6 Conclusions and Future Directions

Here, we give a final summary of the main points of this thesis and discuss briefly possible further research directions.

6.1 Summary of the Contributions

We investigated the problem of efficient pattern matching, especially ReDoS attacks, from both side—from the side of users who want to protect their systems and have their services still available, and from the side of attackers who want to discover vulnerabilities of systems that use regex matchers. We studied the ways of fast matching and the ways of detecting vulnerability of existing regex matchers. We focused on automata-based matchers, especially on those based on online DFA-simulation, which are considered most robust and efficient regex matching engines used nowadays in practice. Namely, we investigated one of the known potential sources of performance problems which are considered regular expressions with bounded repetition.

We presented the first systematic large-scale study of vulnerability of automata-based matching focused on bounded repetition. To this end, we proposed a new ReDoS generator, called `GadgetCA`, that can generate attacks on automata-based matchers used in practice. It is based on the observation that automata-based regex engines generate a part of the state space of DFAs. The algorithm forces them to generate an enormously large state spaces. The study revealed that bounded repetition indeed poses a serious security threat formata-based as well as backtracking matchers. It showed that our generator `GadgetCA` is indeed significantly more successful in creating ReDoS attacks on the given regexes than current state-of-the-at ReDoS generators. We also found that if a regex does not contain bounded repetition, it mostly cannot be used to perform a ReDoS attack on automata-based matchers.

We worked towards improving the matching technique based on online DFA-simulation to handle also regexes with bounded repetition which allows for algorithms running reliably in time linear to the length of the text and independent of the repetition bounds. The general approach is based on first compiling the regexes into nondeterministic CAs and then its determinisation. The main problem is to find a succinct deterministic representation that can perform fast matching linear to the length of the text and independent of repetition bounds.

We made the first step towards succinct determinisation of CAs to deterministic CAs based on generalized subset construction. Our algorithm can produce deterministic CAs exponentially more succinct than the corresponding DFAs. We also developed a simplified and faster version of the general algorithm for the sub-class of so-called monadic CAs. This class is of particular practical relevance since we discovered that most of the

regular expressions with bounded repetition used in practice are of this form. The worst-case complexity of the specialised algorithm is only polynomial in the maximum values of repetition bounds (in contrast to the exponential naive construction). The experiments confirmed that our algorithm produces significantly smaller automata and mitigates the risk of the state space explosion causing a complete failure of determinisation.

The main contribution of this thesis was obtained when we elaborated the determinisation using the idea representing many counters with counting sets. We proposed succinct transformation of a CA into a deterministic CsA, an automaton with a special type of registers which can hold a set of integer values, so-called counting sets. The counting sets support a limited selection of simple set operations that can be implemented to run in constant time regardless of the size of the set. Thus the algorithm produces CsA whose size is independent of the counter bounds and the matching is linear to the length of the input text. Moreover, we proposed a novel compilation of regexes to CAs which generalizes the Antimirov's derivative construction. It is cheap and produces automata without ϵ-transitions whose size is independent of the repetition bounds and linear in the size of the regex. We have implemented the matching algorithm based on CsA simulation into a prototype tool `Chipmunk`. We conducted an extensive experimental evaluation. We targeted a wide range of regexes with bounded repetition coming from various real-life applications. We compared the speed of matching of individual matching engines, namely, `grep`, `RE2`, `SRM`, and `.NET`. We found that `Chipmunk` is much more robust, outperforms the state-of-the-art matchers on regexes with bounded repetition and is not dependent on the size of repetition bounds. It easily solves most of cases in which the existing matchers struggle due to bounded repetition.

6.2 Further Directions

There is a number of interesting directions of further work. We have already started to work in some of these directions. We intend to explore the limits of the idea of counting sets to enlarge and clearly delimit the class of regexes and counting automata that can be succinctly determinised while preserving fast matching. We also plan to explore possible usage of CsAs as a replacement of classical automata in other applications where automata are used, for instance, as symbolic representations of state spaces. For this, we intend to develop CsA counterparts of essential automata techniques, such as Boolean operations and minimization, size-reduction techniques or language emptiness. In case of ReDoS detectors, we see a possibility to further specialize detection of ReDoS attack for specific matchers.

Bibliography

[1] Perl-perlrecharclass. https://perldoc.perl.org/perlrecharclass, 2022.

[2] Parosh Aziz Abdulla, Pavel Krčál, and Wang Yi. R-Automata. In *CONCUR'08*, volume 5201 of *LNCS*, pages 67–81. Springer, 2008.

[3] Alfred V. Aho. Chapter 5 - Algorithms for finding patterns in strings. In *Algorithms and Complexity*, Handbook of Theoretical Computer Science, pages 255–300. Elsevier, 1990.

[4] Valentin Antimirov. Partial derivatives of regular expressions and finite automaton constructions. *Theoretical Computer Science*, 155(2):291 – 319, 1996.

[5] Standards Associations. IEEE Standard for information technology–Portable operating system interface (POSIX(TM)) base specifications. *IEEE Std 1003.1-2017 (Revision of IEEE Std 1003.1-2008)*, pages 1–3951, 2018.

[6] Adam Baldwin. Regular expression denial of service affecting Express.js. https://medium.com/node-security/regular-expression-denial-of-service-affecting-express-js-9c397c164c43, 2016.

[7] Sébastien Bardin, Alain Finkel, Jérôme Leroux, and Laure Petrucci. FAST: Acceleration from theory to practice. *STTT*, 10(5):401–424, 2008.

[8] Gerard Berry and Ravi Sethi. From regular expressions to deterministic automata. *Theoretical Computer Science*, 48(3):117–126, 1986.

[9] Jean Berstel and Jean-Éric Pin. Local languages and the Berry-Sethi algorithm. *Theoretical Computer Science*, 155(2):439–446, 1996.

[10] Henrik Björklund, Wim Martens, and Thomas Timm. Efficient incremental evaluation of succinct regular expressions. In *CIKM'15*, pages 1541–1550. ACM, 2015.

[11] James Britt and Neurogami Secret Laboratory. Regexp - Ruby. https://ruby-doc.org/core-2.3.1/Regexp.html, 2021.

[12] Anne Brüggemann-Klein and Derick Wood. One-unambiguous regular languages. *Information and Computation*, 140(2):229–253, 1998.

[13] Janusz A. Brzozowski. *Derivatives of Regular Expressions*, volume 11, pages 481–494. 1964.

[14] Zhengjun Cao and Lihua Liu. A fast string matching algorithm based on lowlight characters in the pattern. *CoRR*, abs/1401.7110, 2014.

[15] Pascal Caron, Jean-Marc Champarnaud, and Ludovic Mignot. Partial derivatives of an extended regular expression. In *LATA'11*, volume 6638 of *LNCS*, pages 179–191. Springer, 2011.

[16] Jean-Marc Champarnaud and Djelloul Ziadi. Computing the equation automaton of a regular expression in space and time. In *CPM'01*, volume 2089 of *LNCS*, pages 157–168. Springer, 2001.

[17] Haiming Chen and Ping Lu. Checking determinism of regular expressions with counting. *Information and Computation*, 241:302 – 320, 2015.

[18] CNN.com. 1963: The debut of ASCII. `http://edition.cnn.com/TECH/computing/9907/06/1963.idg/`, 2022.

[19] Wikipedia contributors. Regular expression—Wikipedia. `https://en.wikipedia.org/w/index.php?title=Regular_expression&%20oldid=852858998`, 2019.

[20] Intel Corporation. Hyperscan. `https://github.com/intel/hyperscan`, 2021.

[21] Oracle Corporation. RegExp - `JavaScript`. `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/RegExp`, 2021.

[22] Russ Cox. Regular expression matching in the wild. `https://swtch.com/~rsc/regexp/regexp3.html`, 2010.

[23] Loris D'Antoni and Margus Veanes. Minimization of symbolic automata. In *POPL'14*, pages 541–553. ACM, 2014.

[24] Loris D'Antoni and Margus Veanes. Automata modulo theories. *Commun. ACM*, 64(5):86–95, 2021.

[25] James C. Davis. Rethinking regex engines to address ReDoS. In *ESEC/FSE'19*, pages 1256–1258. ACM, 2019.

[26] 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*. PhD thesis, Virginia Polytechnic Institute and State University, 2018.

[27] James C. Davis, Louis G. Michael IV, Christy A. Coghlan, Francisco Servant, and Dongyoon Lee. Why aren't regular expressions a lingua franca? An empirical study on the re-use and portability of regular expressions. In *FSE'19*, pages 443–454. ACM, 2019.

[28] Mark Davis. Unicode nearing 50% of the web. `https://googleblog.blogspot.com/2010/01/unicode-nearing-50-of-web.html`, 2010.

[29] Davis L. Dill, Alan J. Hu, and Howard Wong-Toi. Checking for language inclusion using simulation preorders. In *CAV'91*, volume 575 of *LNCS*. Springer, 1992.

[30] MDN Web Docs. Class pattern - Java. https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/regex/Pattern.html, 2021.

[31] docs.rs. Regex - Rust. https://docs.rs/regex/1.5.4/regex/, 2021.

[32] Mike Haertel et al. GNU grep. https://www.gnu.org/software/grep/, 2022.

[33] Stack Exchange. Outage postmortem. http://stackstatus.net/post/147710624694/outage-postmortem-july-20-2016, 2016.

[34] Sebastian Fischer, Frank Huch, and Thomas Wilke. A play on regular expressions: Functional pearl. *SIGPLAN Not.*, 45(9):357–368, 2010.

[35] Open Information Security Foundation. Suricata. https://suricata.io/, 2022.

[36] Python Software Foundation. RE - Python. https://docs.python.org/3.6/library/re.html, 2021.

[37] Dominik D. Freydenberger. Extended regular expressions: Succinctness and decidability. *Theory Comput. Syst.*, 53(2):159–193, 2013.

[38] Wouter Gelade, Marc Gyssens, and Wim Martens. Regular expressions with counting: Weak versus strong determinism. *SIAM J. Comput.*, 41(1):160–190, 2012.

[39] Rob Glabbeek and Bas Ploeger. Five determinisation algorithms. In *Implementation and Applications of Automata*, pages 161–170. Springer, 2008.

[40] Victor Mikhailovich Glushkov. The abstract theory of automata. *Russian Math. Surveys*, 16:1–53, 1961.

[41] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. Automated whitebox fuzz testing. In *NDSS'08*. The Internet Society, 2008.

[42] Google. RE2. https://github.com/google/re2, 2022.

[43] Saul Gorn. American standard code for information interchange. *Commun. ACM*, 6(8):422–426, 1963.

[44] John Graham-Cumming. Details of the Cloudflare outage on July 2, 2019. https://blog.cloudflare.com/details-of-the-cloudflare-outage-on-july-2-2019/, 2019.

[45] The PHP Group. PCRE patterns - PHP. https://www.php.net/manual/en/regexp.introduction.php, 2021.

[46] Mike Haertel. Why GNU grep is fast. https://lists.freebsd.org/pipermail/freebsd-current/2010-August/019310.html, 2010.

[47] Philip Hazel. PCRE - Perl compatible regular expressions. https://www.pcre.org/, 2022.

[48] Jesper G. Henriksen, Jakob L. Jensen, Michael E. Jørgensen, Nils Klarlund, Robert Paige, Theis Rauhe, and Anders Sandholm. Mona: Monadic second-order logic in practice. In *TACAS'95*, volume 1019 of *LNCS*. Springer, 1995.

[49] Lukáš Holík, Ondřej Lengál, Olli Saarikivi, Lenka Turoňová, Margus Veanes, and Tomáš Vojnar. Succinct determinisation of counting automata via sphere construction. In *APLAS'19*, volume 11893 of *LNCS*, pages 468–489. Springer, 2019.

[50] Lukáš Holík, Ondřej Lengál, Olli Saarikivi, Lenka Turoňová, Margus Veanes, and Tomáš Vojnar. Dataset for the OOPSLA'20 paper "Regex Matching with Counting-Set Automata", 2020.

[51] Dag Hovland. Regular expressions with numerical constraints and automata with counters. In *ICTAC'09*, volume 5684 of *LNCS*, pages 231–245. Springer, 2009.

[52] Dag Hovland. The membership problem for regular expressions with unordered concatenation and numerical constraints. In Adrian-Horia Dediu and Carlos Martín-Vide, editors, *LATA'12*, volume 7183 of *LNCS*, pages 313–324. Springer, 2012.

[53] HubSpot. How Unicode works. https://deliciousbrains.com/how-unicode-works/, 2021.

[54] Intel. Hyperscan 5.4 developer's reference guide, performance considerations. http://intel.github.io/hyperscan/dev-reference/performance.html, 2021.

[55] Pekka Kilpeläinen and Rauno Tuhkanen. Regular expressions with numerical occurrence indicators - preliminary results. In *SPLST'03*, pages 163–173. University of Kuopio, 2003.

[56] Pekka Kilpeläinen and Rauno Tuhkanen. One-unambiguity of regular expressions with numeric occurrence indicators. *Information and Computation*, 205(6):890–916, 2007.

[57] James Kirrage, Asiri Rathnayake, and Hayo Thielecke. Static analysis for regular expression denial-of-service attacks. In *NSS'13*, volume 7873 of *LNCS*, pages 135–148. Springer, 2013.

[58] Stephen Cole Kleene. Representation of events in nerve nets and finite automata. In *Automata studie*, pages 3–14, 1951.

[59] J. Jenny Li and W. Eric Wong. Automatic test generation from communicating extended finite state machine (CEFSM)-Based models. In *ISORC'02*, pages 181–188. IEEE Computer Society, 2002.

[60] Yinxi Liu, Mingxue Zhang, and Wei Meng. Revealer: Detecting and exploiting regular expression denial-of-service vulnerabilities. In *SP'21*, pages 1468–1484. IEEE, 2021.

[61] LLVM project. libFuzzer: A library for coverage-guided fuzz testing. `https://llvm.org/docs/LibFuzzer.html`, 2022.

[62] Blake Loring, Duncan Mitchell, and Johannes Kinder. Sound regular expression semantics for dynamic symbolic execution of JavaScript. In *PLDI'19*, pages 425–438. ACM, 2019.

[63] Martin Roesch et al. Snort: A network intrusion detection and prevention system. `http://www.snort.org`, 2022.

[64] Robert McNaughton and Hisao Yamada. Regular expressions and state graphs for automata. *IRE Trans. Electron. Comput.*, 9(1):39–47, 1960.

[65] Microsoft. Regex.Match method. `https://docs.microsoft.com/en-us/dotnet/api/system.text.regularexpressions.regex.match`, 2020.

[66] Microsoft. CredScan. `https://secdevtools.azurewebsites.net/helpcredscan.html`, 2021.

[67] Microsoft Automata library. Automata and Transducer Library for .NET. `https://github.com/AutomataDotNet/Automata`, 2022.

[68] Lenka Turoňová and Lukáš Holík. Towards smaller invariants for proving coverability. In *EUROCAST'17*, volume 10672 of *LNCS*, pages 109–116. Springer, 2017.

[69] NVIDIA. Nvidia bluefield-2 dpu. `https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/documents/datasheet-nvidia-bluefield-2-dpu.pdf`, 2020.

[70] NVIDIA. Data plane development kit (dpdk). `https://developer.nvidia.com/networking/dpdk`, 2022.

[71] Stack Overflow. Question and answer site for programmers. `http://stackoverflow.com/`, 2022.

[72] OWASP. Regular expression denial of service — ReDoS. `https://owasp.org/www-community/attacks/Regular_expression_Denial_of_Service_-_ReDoS`, 2020.

[73] Scott Owens, John Reppy, and Aaron Turon. Regular-expression derivatives re-examined. In *Journal of Functional Programming*, volume 19, page 173–190. Cambridge University Press, 2009.

[74] Theofilos Petsios, Jason Zhao, Angelos D. Keromytis, and Suman Jana. Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities. In *CCS'17*, pages 2155–2168. ACM, 2017.

[75] Mono project. Mono. https://www.mono-project.com/, 2022.

[76] Asiri Rathnayake. *Semantics, analysis and security of backtracking regular expression matchers*. PhD thesis, University of Birmingham, UK, 2015.

[77] Asiri Rathnayake and Hayo Thielecke. Static analysis for regular expression exponential runtime via substructural logics. *CoRR*, abs/1405.7058, 2014.

[78] RegExLib.com. The internet's first regular expression library. http://regexlib.com/, 2022.

[79] Ondřej Lengál, Jiří Šimáček, and Tomáš Vojnar. VATA: A library for efficient manipulation of non-deterministic tree automata. In *TACAS'12*, volume 7214 of *LNCS*. Springer, 2012.

[80] Robin Sommer et al. The Bro network security monitor. http://www.bro.org, 2022.

[81] Arnaud Rosier, Anita Burgun, and Philippe Mabo. Using regular expressions to extract information on pacemaker implantation procedures from clinical reports. In *AMIA'08*. AMIA, 2008.

[82] Olli Saarikivi, Margus Veanes, Tiki Wan, and Eric Xu. Symbolic regex matcher. In *TACAS'19*, volume 11427 of *LNCS*, pages 372–378. Springer, 2019.

[83] Yuju Shen, Yanyan Jiang, Chang Xu, Ping Yu, Xiaoxing Ma, and Jian Lu. ReScue: Crafting regular expression DoS attacks. In *ASE'18*, pages 225–235. ACM, 2018.

[84] Thomas R. Shiple, James H. Kukula, and Rajeev K. Ranjan. A comparison of Presburger engines for EFSM reachability. In *CAV'98*, volume 1427 of *LNCS*, pages 280–292. Springer, 1998.

[85] Michael Sipser. *Introduction to Theory of Computation*, volume 2. Thomson Course Technology Boston, 2006.

[86] Brad Smith, William D. Harms, Stephanie Burres, Holly Korda, Howard Rosen, and Jamie Davis. Enhancing behavioral health treatment and crisis management through mobile ecological momentary assessment and SMS messaging. *Health Informatics J.*, 18(4):294–308, 2012.

[87] Brad Smith, William D. Harms, Stephanie Burres, Holly Korda, Howard Rosen, and Jamie Davis. Software model checking for people who love automata. In *CAV'13*, volume 8044 of *LNCS*, pages 36–52. Springer, 2013.

[88] Randy Smith, Cristian Estan, and Somesh Jha. XFA: Faster signature matching with extended automata. In *SSP'08*. IEEE, 2008.

[89] Randy Smith, Cristian Estan, Somesh Jha, and Ida Siahaan. Fast signature matching using extended finite automaton (XFA). In *ICISS'08*, volume 5352 of *LNCS*, pages 158–172. Springer, 2008.

[90] snyk.io. ReDoS vulnerabilities in npm spikes by 143% and XSS continues to grow. https://snyk.io/blog/redos-vulnerabilities-in-npm-spikes-by-143-and-xss-continues-to-grow/, 2019.

[91] Henry Spencer. A regular-expression matcher. In *Software Solutions in C*, pages 35–71. Academic Press Professional, Inc., 1994.

[92] Michael Sperberg-McQueen. Notes on finite state automata with counters. https://www.w3.org/XML/2004/05/msm-cfa.html, 2018.

[93] Cristian-Alexandru Staicu and Michael Pradel. Freezing the web: A study of ReDoS vulnerabilities in JavaScript-based web servers. In *USENIX Security'18*, pages 361–376. USENIX Association, 2018.

[94] Satoshi Sugiyama and Yasuhiko Minamide. Checking time linearity of regular expression matching based on backtracking. *IPSJ Online Transactions*, 7:82–92, 2014.

[95] The Sagan team. The Sagan log analysis engine. https://quadrantsec.com/sagan_log_analysis_engine/, 2022.

[96] Iain Truskett. Perl regular expressions reference - Perl. https://perldoc.perl.org/5.22.0/perlreref, 2021.

[97] TrustPort. World class cyber security | TrustPort. https://www.trustport.com/, 2021.

[98] Lenka Turoňová, Lukáš Holík, Ondřej Lengál, Olli Saarikivi, Margus Veanes, and Tomáš Vojnar. Regex matching with counting-set automata. *Proc. ACM Program. Lang.*, 4(OOPSLA):218:1–218:30, 2020.

[99] Lenka Turoňová, Lukáš Holík, Ondřej Lengál, Olli Saarikivi, Margus Veanes, and Tomáš Vojnar. Automata library. https://pajda.fit.vutbr.cz/ituronova/countingautomata, 2022.

[100] Lenka Turoňová, Lukáš Holík, Ondřej Lengál, Margus Veanes, and Tomáš Vojnar. Counting in regexes considered harmful. In *USENIX Security'22*. USENIX Association, 2022.

[101] Petr Janků and Lenka Turoňová. Solving string constraints with approximate parikh image. In *EUROCAST'19*, volume 12013 of *LNCS*, pages 491–498. Springer, 2019.

[102] Milan Češka, Vojtěch Havlena, Lukáš Holík, Ondřej Lengál, and Tomáš Vojnar. Approximate reduction of finite automata for high-speed network intrusion detection. In *TACAS'18*, volume 10806 of *LNCS*. Springer, 2018.

[103] Margus Veanes, Peli de Halleux, and Nikolai Tillmann. Rex: Symbolic regular expression explorer. In *ICST'10*, pages 498–507, 2010.

[104] w3techs.com. Usage of character encodings broken down by ranking. https://w3techs.com/technologies/cross/character_encoding/ranking, 2022.

[105] WebFocus. https://webfocusinfocenter.informationbuilders.com/wfappent/TL2s/TL_nls/source/01_nls_7720.htm, 2022.

[106] Nicolaas Weideman. *Static analysis of regular expressions*. PhD thesis, University of Stellenbosch, 2007.

[107] Nicolaas Weideman. RegexStatic. https://github.com/NicolaasWeideman/RegexStaticAnalysis, 2015.

[108] 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 *CIAA'16*, volume 9705 of *LNCS*, pages 322–334. Springer, 2016.

[109] Valentin Wüstholz, Oswaldo Olivo, Marijn J. H. Heule, and Isil Dillig. Static detection of DoS vulnerabilities in programs that use regular expressions. In *TACAS'17*, volume 10206 of *LNCS*, pages 3–20, 2017.

[110] Liu Yang, Rezwana Karim, Vinod Ganapathy, and Randy Smith. Improving NFA-based signature matching using ordered binary decision diagrams. In *RAID'10*, pages 58–78. Springer, 2010.