Towards Efficient Matching of Regexes with Backreferences using Register Set Automata

VOJTĚCH HAVLENA, Brno University of Technology, Czech Republic
LUKÁŠ HOLÍK, Brno University of Technology, Czech Republic and Aalborg University, Denmark
ONDŘEJ LENGÁL, Brno University of Technology, Czech Republic
JAN VAŠÁK, Brno University of Technology, Czech Republic
SABÍNA GULČÍKOVÁ, Brno University of Technology, Czech Republic

Matching regexes (regular expressions) is a common problem in many areas of computer science, with requirements on high speed and robust performance. Regexes with backreferences allow one to express certain patterns (even beyond regular) concisely, however, since the matching is usually done by backtracking, the matching speed can degrade to a degree that constitutes a service failure or a security threat. To facilitate high-speed matching of such regexes, we propose *register set automata* (RSAs), an extension of register automata where registers can contain *sets* of symbols (from a potentially infinite alphabet) and the following operations are supported: adding input values to registers, merging or clearing registers, and testing whether a register contains a value. We show that a large class of *register automata* can be transformed into *deterministic* RSAs, which can serve as a basis for fast matching of a family of regexes with single-letter capture groups and backreferences. We also give a derivative-based algorithm for transforming a large class of regexes with backreferences to register automata and show that the time complexity of matching is linear and quadratic to the length of the input for finite and infinite alphabets respectively. Our prototype implementation of a regex matcher shows that our approach can significantly improve the robustness of state-of-the-art regex matchers on regexes with backreferences. We also study the theoretical properties of the model and show that the emptiness problem for RSAs is decidable and complete for the $\mathbf{F}_\omega$ class and that RSAs are incomparable in expressive power to other popular automata models over data words.

CCS Concepts: • **Security and privacy** → **Denial-of-service attacks**; • **Theory of computation** → **Automata over infinite objects**; **Regular languages**; • **Applied computing** → *Document searching*.

Additional Key Words and Phrases: regular expression matching, backreferences, ReDoS, determinization, Antimirov's derivatives, register automata, register set automata

ACM Reference Format:
Vojtěch Havlena, Lukáš Holík, Ondřej Lengál, Jan Vašák, and Sabína Gulčíková. 2026. Towards Efficient Matching of Regexes with Backreferences using Register Set Automata. *Proc. ACM Program. Lang.* 10, PLDI, Article 203 (June 2026), 26 pages. https://doi.org/10.1145/3808281

1 Introduction

Regular expression (regex) matching is a task performed routinely in computer programs, such as in searching, data validation, parsing, finding and replacing, data leak detection, or syntax highlighting. Studies show that 30–40 % of Java, JavaScript, and Python software use regex matching [15]. In

Authors' Contact Information: Vojtěch Havlena, Brno University of Technology, Brno, Czech Republic, ihavlena@fit.vutbr.cz; Lukáš Holík, Brno University of Technology, Brno, Czech Republic and Aalborg University, Aalborg, Denmark, holik@fit.vutbr.cz; Ondřej Lengál, Brno University of Technology, Brno, Czech Republic, lengal@fit.vutbr.cz; Jan Vašák, Brno University of Technology, Brno, Czech Republic, xvasak01@stud.fit.vutbr.cz; Sabína Gulčíková, Brno University of Technology, Brno, Czech Republic, xgulci00@stud.fit.vutbr.cz.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2026 Copyright held by the owner/author(s).
ACM 2475-1421/2026/6-ART203
https://doi.org/10.1145/3808281

many of these applications, performance of regex matching is crucial, such as in various online services, where underperforming regex matching over user input can cause the server to become unresponsive and unusable. If such a situation is caused intentionally by a malicious user, we talk about the so-called *regular expression denial of service* (ReDoS) attack [1]. ReDoS is a real-world threat; it caused, for instance, the 2016 outage of StackOverflow [22] or rendered vulnerable websites that were using the Express.js framework [4].

While there are matchers that have a reasonably robust performance on basic regexes—such as grep [21], RE2 [28], or HyperScan [73]—their performance on *extended regexes* can quickly deteriorate [68] (or they do not support the particular extensions at all [27, 34]). Examples of such extensions are, e.g., bounded counters [68, 69], lookaheads and lookbehinds, or capture groups and backreferences, which are the topic of this paper. The performance degrades mainly due to the inability of matchers to use the fastest automata-based algorithms that have constant or at most linear per-character cost. These algorithms are based on the automata determinization,[1] which cannot easily accommodate the syntactic features of extended regexes (the language of a regex with backreferences may even not be regular). The matchers then usually fall back to *backtracking*, which is much slower on average and exponential at worst.

Pattern matching using regular expressions with backreferences is performed ubiquitously, e.g., in validating user inputs on web pages, processing text using the `grep` and `sed` tools, transforming XML documents, or detecting network incidents [8, 54]. Consider, for instance, the (extended) regex

$$R_{\texttt{\textbackslash 3\textbackslash 2\textbackslash 1}} = \texttt{/(.).*;.*(.).*;.*(.).*\textbackslash 3\textbackslash 2\textbackslash 1/},$$

where we use the wildcard "`.`" to denote any symbol except the semicolon "`;`" (i.e., it stands for the character class "`[^;]`"), parentheses "`(...)`" to denote the so-called *capture groups*, and "`\x`" to denote a *backreference* to the string captured by the `x`'th capture group (the semicolon "`;`" serves in the regexs as a delimiter). Intuitively, the regex matches input strings w that can be seen as a concatenation of three strings $w = uvz$ such that u has the structure $u = u_1\,;u_2\,;u_3$ with $u_1, u_2, u_3 \in (\Sigma \setminus \{;\})^+$, v is a string of three characters $a_3a_2a_1$ such that $a_i \in u_i$ for $i \in \{1, 2, 3\}$, and $z \in (\Sigma \setminus \{;\})^*$ (such a regex can describe, e.g., a simplified version of the match rule of some XML transformation). Trying (unsuccessfully) to find a match of the regex in the randomly generated 42-character-long string

"`ah;jk2367ash;la5akv45lwkjb9f.dj5fqkbxsfyrf`"

using the state-of-the-art PCRE2 regex matcher available at regex101.com [53] takes 10,374 steps before reporting *no match*.[2] Ideally, the matcher should take only 42 steps, one step for every character in the string. This $\frac{10{,}374}{42} = 247\times$ slowdown is caused by the so-called *catastrophic backtracking*—the PCRE2 matcher is based on backtracking and, since the regex is nondeterministic, the backtracking algorithm needs to try all possibilities of placing the three capture groups before concluding that there is no match. From a theoretical point of view, this inadequate performance is hardly a surprise, since matching of regexes with backreferences is an NP-hard problem [2]. This theoretical obstacle, however, does not need to be a show-stopper, as for many regexes appearing in practice, there is still hope that a matching algorithm with a time complexity linear (or at most quadratic) to the size of the input is possible.

As mentioned above, practical means of avoiding backtracking in such cases are not available since the automata determinization does not support backreferences. Indeed, neither of the two currently most advanced regex matchers in the industry, RE2 [28] and HyperScan [73], supports

[1] In practice, the matchers are based on Thompson's algorithm [66], which does not build the (possibly prohibitively large) DFA *a priori* but, instead, *on the fly*, while using cache to store already constructed parts of the DFA.

[2] Finding a match of the regex `/(.).*(.).*(.).*\3\2\1/` in the same text took 169,379 steps before *no match* was reported. This regex is more challenging than $R_{\texttt{\textbackslash 3\textbackslash 2\textbackslash 1}}$—it does not use delimiters (semicolons in $R_{\texttt{\textbackslash 3\textbackslash 2\textbackslash 1}}$).

backreferences, due to a missing efficient algorithm [27, 34]. The main obstacle is a lack of a suitable deterministic automata model with a fast membership test that would support backreferences.

In this paper, we develop such a formal model, particularly suited for fast matching of a class of these regexes where capture groups are single-letter (the single-letter backreferences constitute a large portion of backreferences in our dataset collected for real-world applications). The formal model are *register set automata* (RSAs), an extension of *register automata* [18, 37] (RAs) where registers can contain sets of symbols instead of just single symbols (as for RAs). Deterministic RSAs (DRSAs) can be simulated in an efficient and robust regex-matching algorithm in time linear to the length of the input text, for finite alphabets, or quadratic to the size of the input for infinite alphabets. Our key result is a *RA to* DRSA *determinization (semi-)algorithm*; we also propose a partial-derivative-based algorithm for compiling a regex with single-letter backreferences to an RA that completes the matching workflow. We implemented a prototype matcher based on these results and experimented with a sample of regexes with single-letter backreferences extracted from a comprehensive real-world benchmark set. Although our determinisation is not complete (it may fail), the experiments show that it is quite reliable, and our matcher indeed significantly improves predictability of matching by state-of-the-art matchers and reduces the danger of ReDoS.

Theory of Register Set Automata. We also deliver a number of positive theoretical results related to our new RSA automata model, which place it into the landscape of automata over infinite alphabets. First, the model strictly generalizes RAs [18, 37] and is incomparable to (one-way) alternating RAs, another popular powerful generalization of RAs [18]. Besides the aforementioned RA to RSA determinization, the core property of RSAs is that their emptiness problem is decidable, although for a higher price than for RAs—the complexity blows from PSPACE-complete for RAs to $\mathbf{F}_\omega$-complete (i.e., Ackermannian) for RSAs.

Importantly, since we can now determinize RAs to RSAs and test emptiness of RSAs, we can use them to test language inclusion of RAs by the standard approach: by determinizing, complementing, and testing emptiness of the intersection (complementing a DRSA is done easily by swapping final and non-final states). Testing RA inclusion is an essential problem in many of their applications, such as in their minimization, learning [10, 26, 36], checking for fixpoint in regular model checking [12], checking XML schema subsumption [67], verification of parameterized concurrent programs with shared memory [35], and is an essential component of the RA toolkit. Unfortunately, inclusion of RAs is in general an undecidable problem [48], which has forced researchers to either find ways to approximate the inclusion test—e.g., via several membership tests [26, 36] or by an abstraction refinement semi-algorithm using interpolation [35]—or restrict themselves to other models with decidable inclusion problem, such as deterministic RAs (whose expressive power is quite limited) [46] or session automata [10]. We note that inclusion of languages described by regexes with backreferences is also undecidable [24].

Contribution. Let us summarise the main contributions of the paper:

(1) Introduction of the model of register set automata and its theoretical analysis: Showing its closure properties, placing the RSA model into the landscape of automata-with-registers models, and proving $\mathbf{F}_\omega$-completeness of the emptiness problem for RSAs by showing interreducibility with the coverability problem for transfer Petri nets, discussing the power of several extensions of the model.
(2) Designing a (semi-)algorithm that can determinize an RA into a DRSA and showing that it is complete for the class of languages that can be obtained from RAs with one register and no disequality tests by Boolean operations (i.e., union, intersection, and complement).
(3) Developing a partial-derivative based algorithm for converting a large class of regexes with backreferences to RAs.

(4) Showing that DRSA-based regex matching can be done in linear or quadratic time, depending on whether the size of the alphabet is finite or infinite respectively.
(5) Experimentally confirming that simulation of the deterministic RSA obtained by our algorithm is a practical matching algorithm for regexes with single-letter backreferences, and that it is significantly more resilient against ReDoS than state-of-the-art matchers.

2 Preliminaries

Sets and Functions. We use $\mathbb{N}$ to denote the set of natural numbers without 0, $\mathbb{N}_0$ to denote $\mathbb{N} \cup \{0\}$, and $[n]$ for $n \in \mathbb{N}$ to denote the set $\{1, \ldots, n\}$ (we note that $[0] = \emptyset$). We sometimes use "$\cdot$" to denote an *ellipsis*, i.e., a value that can be ignored. For a (partial) function $f : A \rightharpoonup B$, a set $C \subseteq B$, and an element $d \in B$, we use $f[C \mapsto d]$ to denote the function $f[C \mapsto d](x) = d$ if $f(x) \in C$ and $f[C \mapsto d](x) = f(x)$ otherwise. Furthermore, for a set $X \subseteq A$, we use $f_{|X}$ to denote the restriction of f to X defined as $f_{|X} = f \cap (X \times B)$.

Data Words. Let us fix a finite nonempty *alphabet* Σ and an infinite *data domain* $\mathbb{D}$. A (finite) *data word* of *length* n is a function $w : [n] \to (\Sigma \times \mathbb{D})$; we use $|w| = n$ to denote its length and $w_1, \ldots, w_n$ to denote its symbols. The *empty word* of length 0 is denoted ϵ. We use $\Sigma[w]$ and $\mathbb{D}[w]$ to denote the *projection* of w onto the respective domain (e.g., if $w = \langle a, 1\rangle\langle b, 2\rangle\langle b, 3\rangle$, then $\Sigma[w] = abc$ and $\mathbb{D}[w] = 123$) and, given $a \in \Sigma$, we use $a[w_i]$ as a shortcut for $\Sigma[w_i] = a$.

Register Automata on Data Words [18, 37]. A (nondeterministic one-way) *register automaton* (on data words), abbreviated as (N)RA, is a tuple $\mathcal{A} = (Q, \mathbf{R}, \Delta, I, F)$ where Q is a finite set of *states*, $\mathbf{R}$ is a finite set of *registers*, $I \subseteq Q$ is a set of *initial states*, $F \subseteq Q$ is a set of *final states*, and $\Delta \subseteq Q \times \Sigma \times 2^{\mathbf{R}} \times 2^{\mathbf{R}} \times (\mathbf{R} \to \mathbf{R} \cup \{in, \bot\}) \times Q$ is a *transition relation* such that if $\tau : (q, a, g^=, g^{\neq}, up, s) \in \Delta$, then $g^= \cap g^{\neq} = \emptyset$. We use $q \xrightarrow{a \mid g^=, g^{\neq}, up} s$ to denote τ (and often drop from up mappings $r \mapsto r$ for $r \in \mathbf{R}$, which we treat as implicit). The semantics of τ is that $\mathcal{A}$ can move from state q to state s if the Σ-symbol at the current position of the input word is a and the $\mathbb{D}$-value at the current position is equal to all registers from $g^=$ and not equal to any register from $g^{\neq}$; the content of the registers is updated so that $r_i \leftarrow up(r_i)$ (i.e., r_i can be assigned the value of some other register, the current $\mathbb{D}$-symbol, denoted by in, or it can be cleared by being assigned $\bot$).

A *configuration* of $\mathcal{A}$ is a pair $c \in Q \times (\mathbf{R} \to \mathbb{D} \cup \{\bot\})$, i.e., it consists of a state and an assignment of data to registers. An *initial configuration* of $\mathcal{A}$ is a pair $c_{init} = (q_{init}, \{r \mapsto \bot \mid r \in \mathbf{R}\})$ with $q_{init} \in I$. Suppose $c_1 = (q_1, f_1)$ and $c_2 = (q_2, f_2)$ are two configurations of $\mathcal{A}$. We say that c_1 can make a *step* to c_2 over $\langle a, d\rangle \in \Sigma \times \mathbb{D}$ using transition $\tau : q \xrightarrow{a \mid g^=, g^{\neq}, up} s \in \Delta$, denoted as $c_1 \vdash_{\tau}^{\langle a,d\rangle} c_2$, iff

(1) $d = f_1(r)$ for all $r \in g^=$,
(2) $d \neq f_1(r)$ for all $r \in g^{\neq}$, and
(3) for all $r \in \mathbf{R}$, we have $f_2(r) = \begin{cases} f_1(r') & \text{if } up(r) = r' \in \mathbf{R}, \\ d & \text{if } up(r) = in, \text{ and} \\ \bot & \text{if } up(r) = \bot. \end{cases}$

A *run* ρ of $\mathcal{A}$ over the word $w = \langle a_1, d_1\rangle \ldots \langle a_n, d_n\rangle$ from a configuration c is a sequence of alternating configurations and transitions $\rho = c_0\tau_1c_1\tau_2 \ldots \tau_nc_n$ such that $\forall 1 \leq i \leq n : c_{i-1} \vdash_{\tau_i}^{\langle a_i,d_i\rangle} c_i$ and $c_0 = c$. We say that ρ is *accepting* if c is an initial configuration, $c_n = (s, f)$, and $s \in F$. The *language* $\mathcal{L}(\mathcal{A})$ accepted by $\mathcal{A}$ is defined as $\mathcal{L}(\mathcal{A}) = \{w \in (\Sigma \times \mathbb{D})^* \mid \mathcal{A} \text{ has an accepting run over } w\}$.

We say that $\mathcal{A}$ is a *deterministic RA* (DRA) if for all states $q \in Q$ and all $a \in \Sigma$, it holds that for any two distinct transitions $q \xrightarrow{a \mid g_1^=, g_1^{\neq}, up_1} s_1, q \xrightarrow{a \mid g_2^=, g_2^{\neq}, up_2} s_2 \in \Delta$ we have that $g_1^= \cap g_2^{\neq} \neq \emptyset$ or $g_2^= \cap g_1^{\neq} \neq \emptyset$. $\mathcal{A}$ is *complete* if for all states $q \in Q$, symbols $a \in \Sigma$, and $g \subseteq \mathbf{R}$, there is a transition $q \xrightarrow{a \mid g^=, g^{\neq}, up} s$ such that $g^= \subseteq g$ and $g \cap g^{\neq} = \emptyset$.

Universal RAs. A *universal RA* (URA) $\mathcal{A}_U$ is defined exactly as an NRA with the exception of its language. The language of $\mathcal{A}_U$ is the set $\mathcal{L}(\mathcal{A}_U) = \{w \in (\Sigma \times \mathbb{D})^* \mid \text{every run of } \mathcal{A}_U \text{ on } w \text{ is accepting}\}$ (we emphasize that if a run cannot continue from some state over the current input symbol, then it is not accepting). The concept of *universality* in the name is linked to the use of universal quantification over runs in deciding acceptance.

There is indeed duality between NRAs and URAs, as stated by the following fact.

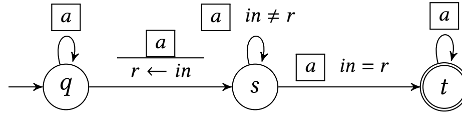
Fact 1. *For every* NRA $\mathcal{A}_N$*, there is a* URA *accepting the complement of* $\mathcal{L}(\mathcal{A}_N)$*. Conversely, for every* URA $\mathcal{A}_U$*, there is an* NRA *accepting the complement of* $\mathcal{L}(\mathcal{A}_U)$*.*

Proof. For both parts, the complement automaton is obtained by (i) adding a rejecting *sink* state to the automaton, (ii) completing the transition relation (i.e., adding transitions for undefined combinations of symbols and guards to the sink state) of the input automaton, and (iii) swapping final and non-final states. □

Example 2.1. Consider the language of words over $\Sigma = \{a\}$ that contain two occurrences of some data value, i.e., the language

$$L_{\exists repeat} = \{w \mid \exists i, j \colon i \neq j \wedge \mathbb{D}[w_i] = \mathbb{D}[w_j]\}.$$

An NRA recognising this language is in the following figure:



Formally, it is an NRA $\mathcal{A} = (\{q, s, t\}, \{r\}, \Delta, \{q\}, \{t\})$ where the transition relation is defined as $\Delta = \{q \xrightarrow{a \mid \emptyset, \emptyset, \emptyset} q, q \xrightarrow{a \mid \emptyset, \emptyset, \{r \mapsto in\}} s, s \xrightarrow{a \mid \emptyset, \{r\}, \emptyset} s, s \xrightarrow{a \mid \{r\}, \emptyset, \emptyset} t, t \xrightarrow{a \mid \emptyset, \emptyset, \emptyset} t\}$ (actually, the guard $in \neq r$ on the self-loop over s is redundant). Recall that $\emptyset$ for an update denotes the mapping $\{r \mapsto r\}$. We note that $L_{\exists repeat}$ is not expressible by any DRA or URA.[3] Intuitively, $\mathcal{A}$ waits in q until it nondeterministically guesses the input data value that should be repeated, stores it into register r, and moves to state s. In state s, it is waiting to see the data value again, upon which it moves to the accepting state t and reads out the rest of the word.

On the other hand, the complement of the language, i.e., the language of words where no two positions have the same data value, formally,

$$L_{\neg\exists repeat} = \{w \mid \forall i, j \colon i \neq j \implies \mathbb{D}[w_i] \neq \mathbb{D}[w_j]\},$$

is not expressible by any DRA or NRA [37, Proposition 5 and its proof], but is expressible by a URA. The URA accepting $L_{\neg\exists repeat}$ looks similar to the NRA $\mathcal{A}$ above with the exception of final states, which are $\{q, s\}$ (note that in URAs, in order to accept a word, all runs over the word need to accept, so in order to accept in this example, all runs of the URA need to avoid the state t). □

We use NRA$^=$, URA$^=$, and DRA$^=$ to denote the sub-classes of NRAs, URAs, and DRAs with *no disequality* guards, i.e., automata where for every transition $q \xrightarrow{a \mid g^=, g^{\neq}, up} s$ it holds that $g^{\neq} = \emptyset$. Furthermore, for a class C of automata with registers and $n \in \mathbb{N}$, we use C_n to denote the sub-class of C containing automata with at most n registers (e.g., DRA$_2$). We abuse notation and use C to also denote the class of languages defined by C.

[3] This can be shown by contradiction, assuming that there is a DRA with k registers accepting $L_{\exists repeat}$. One can then take a run over some rejected word of the length $2k + 2$, cut it in half, and modify the second half of the word to contain some symbol that is in the first half of the word but is not stored any register. Because of determinism, the DRSA should also reject the modified word, but then the accepted language is not $L_{\exists repeat}$. Contradiction.

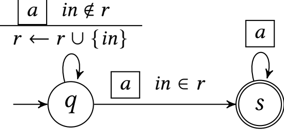


Fig. 1. DRSA_1 for $L_{\exists repeat}$

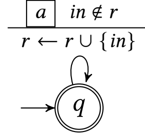


Fig. 2. DRSA_1 for $L_{\neg\exists repeat}$

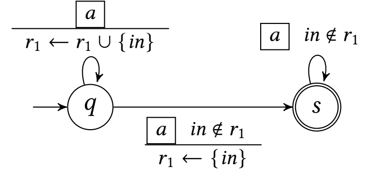


Fig. 3. RSA_1 for $L_{\neg\forall repeat}$

3 Register Set Automata

On this section, we introduce the model of register set automata, which differ from RAs mainly in the ability to store into registers *sets of values* instead of just single data elements. A (nondeterministic) *register set automaton* (on data words), abbreviated as (N)RSA is a tuple $\mathcal{A}_S = (Q, \mathbf{R}, \Delta, I, F)$ where $Q, \mathbf{R}, I, F$ are the same as for RAs and $\Delta \subseteq Q \times \Sigma \times 2^{\mathbf{R}} \times 2^{\mathbf{R}} \times (\mathbf{R} \to 2^{\mathbf{R}\cup\{in\}}) \times Q$ such that if $q \xrightarrow{a \mid g^{\in}, g^{\notin}, up} s \in \Delta$, then $g^{\in} \cap g^{\notin} = \emptyset$ (as with NRAs, we often do not write mappings $r \mapsto \{r\}$ for $r \in \mathbf{R}$ when defining up). The semantics of a transition $q \xrightarrow{a \mid g^{\in}, g^{\notin}, up} s$ is that $\mathcal{A}_S$ can move from state q to state s if the Σ-symbol at the current position of the input word is a and the $\mathbb{D}$-value at the current position is in all registers from $g^{\in}$ and in no register from $g^{\notin}$; the content of the registers is updated so that $r_i \leftarrow \bigcup\{x \mid x \in up(r_i)\}$ (i.e., r_i can be assigned the union of values of several registers, possibly including the current $\mathbb{D}$-symbol denoted by in).

A *configuration* of $\mathcal{A}_S$ is a pair $c \in Q \times (\mathbf{R} \to 2^{\mathbb{D}})$, i.e., it consists of a state and an assignment of sets of data values to registers. An *initial configuration* of $\mathcal{A}_S$ is a pair $c_{init} = (q_{init}, \{r \mapsto \emptyset \mid r \in \mathbf{R}\})$ with $q_{init} \in I$. Let $c_1 = (q_1, f_1)$ and $c_2 = (q_2, f_2)$ be two configurations of $\mathcal{A}_S$. We say that c_1 can make a *step* to c_2 over $\langle a, d\rangle \in \Sigma \times \mathbb{D}$ using transition $t\colon q \xrightarrow{a \mid g^{\in}, g^{\notin}, up} s \in \Delta$, denoted as $c_1 \vdash_t^{\langle a,d\rangle} c_2$, iff

(1) $d \in f_1(r)$ for all $r \in g^{\in}$,
(2) $d \notin f_1(r)$ for all $r \in g^{\notin}$, and
(3) for all $r \in \mathbf{R}$, we have $f_2(r) = \bigcup\{f_1(r') \mid r' \in \mathbf{R}, r' \in up(r)\} \cup \begin{cases}\{d\} & \text{if } in \in up(r) \text{ and} \\ \emptyset & \text{otherwise.}\end{cases}$

The definition of a run and language of $\mathcal{A}_S$ is then the same as for NRAs.

We say that the RSA $\mathcal{A}_S$ is *deterministic* (DRSA) if for all states $q \in Q$ and all $a \in \Sigma$, it holds that for any two distinct transitions $q \xrightarrow{a \mid g_1^{\in}, g_1^{\notin}, up_1} s_1, q \xrightarrow{a \mid g_2^{\in}, g_2^{\notin}, up_2} s_2 \in \Delta$ we have that $g_1^{\in} \cap g_2^{\notin} \neq \emptyset$ or $g_2^{\in} \cap g_1^{\notin} \neq \emptyset$.

Example 3.1. A DRSA accepting the language $L_{\exists repeat}$ from Example 2.1 is in Fig. 1. Formally, it is a DRSA_1 $\mathcal{A} = (\{q, s\}, \{r\}, \Delta, \{q\}, \{s\})$ where $\Delta = \{q \xrightarrow{a \mid \emptyset, \{r\}, \{r \mapsto \{r, in\}\}} q, q \xrightarrow{a \mid \{r\}, \emptyset, \emptyset} s, s \xrightarrow{a \mid \emptyset, \emptyset, \emptyset} s\}$. Intuitively, the DRSA waits in q and accumulates the so-far seen input data values in register r (we use $r \leftarrow r \cup \{in\}$ to denote the update $r \mapsto \{r, in\}$). Once the DRSA reads a value that is already in r, it moves to s and accepts. □

Example 3.2. A DRSA_1 accepting the language $L_{\neg\exists repeat}$ from Example 2.1 is in Fig. 2. Intuitively, the automaton stays in state q and accumulates input data values in register r, making sure the input data value has not been seen previously. □

Example 3.3. Consider the following language:

$$L_{\neg\forall repeat} = \{w \mid \exists i \forall j\colon i \neq j \implies \mathbb{D}[w_i] \neq \mathbb{D}[w_j]\}$$

Intuitively, $L_{\neg\forall repeat}$ is the language of all words containing a data value with exactly one occurrence. This language is accepted, e.g., by the RSA_1 in Fig. 3. The RSA stays in state q, collecting the seen values into its register, and at some point, when it sees a value not seen previously, it

nondeterministically moves to s, remembering the value in its register. Then, at state s, the RSA just checks that it does not see the remembered value any more. We note that $L_{\neg\forall repeat}$ cannot be accepted by a DRSA (cf. the proof of Theorem 4.6). □

4 Properties of Register Set Automata

In this section, we establish decidability and complexity of basic decision problems for RSAs and their closure properties. First, we claim that RSAs generalise NRAs.

FACT 2. *For every $n \in \mathbb{N}$ and NRA_n, there exists an RSA_n accepting the same language.*

The next theorem shows the core property of RSAs: that their emptiness problem is decidable, however, for a much higher price than for NRAs, for which it is PSPACE-complete[4] [18]. For classifying the complexity of the problem, we use the hierarchy of fast-growing complexity classes of Schmitz [57], in particular the class $\mathbf{F}_\omega$, which, intuitively, corresponds to Ackermannian problems closed under primitive-recursive reductions.

THEOREM 4.1. *The emptiness problem for RSA is decidable, in particular, $\mathbf{F}_\omega$-complete.*

SKETCH OF PROOF. The proof is done by showing interreducibility of RSA emptiness with coverability in *transfer Petri nets* (TPNs) (often used for modelling the so-called *broadcast protocols*), which is a known $\mathbf{F}_\omega$-complete problem [58–60]. In the following, we briefly describe both directions of the reduction (see [30] for details and examples).

(RSA emptiness ≤ TPN coverability) Intuitively, the conversion of an RSA $\mathcal{A} = (Q, \mathbf{R}, \Delta, I, F)$ into a TPN $\mathcal{N}_\mathcal{A}$ is done in the following way. The set of places of $\mathcal{N}_\mathcal{A}$ will be as follows: (i) one place for each state of $\mathcal{A}$, (ii) two special places *init* and *fin*, and (iii) one place for every subset $rgn \subseteq \mathbf{R}$; these places are used to represent all possible intersections of values held in registers. E.g., if there are four tokens in the place representing $r_1 \cap r_2$, it means that there are exactly four different data values stored in both r_1 and r_2 and in no other register. Each transition t of $\mathcal{A}$ is simulated by one or more TPN transitions between places representing its source and target states. The number of respective TPN transitions depends on how specific the guard is in the original automaton, since we need to distinguish every possible option of *in* being in some region $rgn \in 2^\mathbf{R}$. The transitions move the token between the places corresponding to t's source and target states and, moreover, use the *broadcast* arcs to move tokens between the places representing regions, according to the manipulation of the set-registers in the update function of t. The special place *init* is used to have a single starting marking (it nondeterministically chooses one state from I) and the place *fin* is used as the coverability test target; all places corresponding to final states of $\mathcal{A}$ can simply transition into it.

(TPN coverability ≤ RSA emptiness) Given a TPN $\mathcal{N}$, the RSA $\mathcal{A}_\mathcal{N}$ simulating it will have the following structure. There will be a state q_{main}, which will be active before and after the simulation of firing each transition of $\mathcal{N}$. Moreover, there will be one register for every place of $\mathcal{N}$; individual tokens in the places will be simulated by unique data values from $\mathbb{D}$ stored in the corresponding registers. For each transition of $\mathcal{N}$, there will be a *gadget*, doing a cycle on q_{main}, that represents the semantics of $\mathcal{N}$'s transition. Each such gadget is composed of several *protogadgets*, which simulate basic actions performed during the transition (adding a token to a place, removing a token, moving all tokens between places). Implementation of adding a token and moving tokens is relatively easy, the tricky part is removing a token, since RSAs do not support removing a data value from a register. We solve this by using

[4] Note that for an alternative definition of NRAs considered in [37, 55], where no two registers can contain the same data value, the problem is NP-complete [55].

a *lossy remove*: i.e., if *one* token is to be removed from a place, we simulate it by removing *at least one* token (but potentially more). This will not preserve *reachability*, but it is enough to preserve *coverability*. Moreover, there will also be an *initial* part setting the contents of the registers to reflect the initial marking of $\mathcal{N}$ (terminating in q_{main}) and a *final* part that checks the coverability by removing (again in a lossy way) tokens from places, terminating in a single final state. □

Remark 1. *Since* RSA*s generalise* NRA*s, their universality, equivalence, and language inclusion problems are all undecidable.*

4.1 Closure Properties

The closure properties of RSAs under Boolean operations are the same as for NRAs (cf. [37, Proposition 5, Theorem 3]).

THEOREM 4.2. *The following closure properties hold for the class* RSA*:*

(1) RSA *is closed under union and intersection.*
(2) RSA *is not closed under complement.*

SKETCH OF PROOF. The proofs for closure under union and intersection are standard. For showing the non-closure under complement, consider the language $L_{\neg\forall repeat}$ from Example 3.3, which can be accepted by RSA. We use a similar technique as in the proof of Proposition 3.2 in [23] and show that if there were an RSA accepting its complement, namely, the language

$$L_{\forall repeat} = \{w \mid \forall i \exists j \colon i \neq j \land \mathbb{D}[w_i] = \mathbb{D}[w_j]\},$$

RSAs could be used to decide the emptiness problem of a Minsky machine, which is a known undecidable problem. See the full proof in [30] for more details. □

For RSAs with a limited number of registers, we lose the closure under intersection.

THEOREM 4.3. *For each $n \in \mathbb{N}$, the class* RSA_n *is closed under union and not closed under intersection and complement.*

SKETCH OF PROOF. The proof of closure of RSA_n under union is standard. For proving non-closure of RSA_1 under intersection, we consider the two following RSA_1 languages

$$\mathcal{L}_1^A = \{w \mid \mathbb{D}[w_1] = \mathbb{D}[w_{|w|}]\} \qquad \text{and} \qquad \mathcal{L}_1^B = \{w \mid \mathbb{D}[w_2] = \mathbb{D}[w_{|w|-1}]\} \tag{1}$$

and show that their intersection cannot be accepted by RSA_1. This argument can be extended to RSA_n for $n > 1$. Non-closure of RSA_n under complement then follows from De Morgan's laws. □

THEOREM 4.4. DRSA *is closed under union, intersection, and complement.*

PROOF. The proofs are standard (product construction and swapping (non)-final states). □

THEOREM 4.5. *For each $n \in \mathbb{N}$, the class* DRSA_n *is closed under complement and not closed under union and intersection.*

PROOF. Proof of closure under complement is standard. Non-closure under intersection is done similarly as in the proof of Theorem 4.3. Non-closure for union follows from De Morgan's laws. □

As with RAs, nondeterminism also allows bigger expressivity for RSAs.

THEOREM 4.6. DRSA $\subsetneq$ RSA

Algorithm 1: Determinization of an NRA into a DRSA

Input : Copyless NRA $\mathcal{A} = (Q, \mathbf{R}, \Delta, I, F)$
Output: DRSA $\mathcal{A}' = (Q', \mathbf{R}, \Delta', I', F')$ with $\mathcal{L}(\mathcal{A}') = \mathcal{L}(\mathcal{A})$ or $\bot$

1 $Q' \leftarrow \text{worklist} \leftarrow I' \leftarrow \{(I, c_0 = \{r \mapsto 0 \mid r \in \mathbf{R}\})\}$;
2 $\Delta' \leftarrow \emptyset$;
3 **while** worklist $\neq \emptyset$ **do**
4 $\quad (S, c) \leftarrow \text{worklist}.pop()$;
5 $\quad$ **foreach** $a \in \Sigma, g \subseteq \{r \in \mathbf{R}[S] \mid c(r) \neq 0\}$ **do**
6 $\quad\quad T \leftarrow \left\{q \xrightarrow{a \mid g^{=}, g^{\neq}, \cdot} q' \in \Delta \mid q \in S, g^{=} \subseteq g, g^{\neq} \cap g = \emptyset\right\}$;
7 $\quad\quad S' \leftarrow \left\{q' \mid \cdot \xrightarrow{\cdot \mid \cdot, \cdot, \cdot} q' \in T\right\}$;
8 $\quad\quad$ **if** $\exists q \xrightarrow{\cdot \mid \cdot, g^{\neq}, \cdot} q' \in T, \exists r \in g^{\neq} : c(r) > 1$ **then return** $\bot$;
9 $\quad\quad T^{\bullet} = \{q \xrightarrow{a \mid g^{=}, g^{\neq}, up[g^{=} \mapsto in]} q' \mid q \xrightarrow{a \mid g^{=}, g^{\neq}, up} q' \in T\}$;
10 $\quad\quad$ **foreach** $r_i \in \mathbf{R}$ **do** `// update the register size classes`
11 $\quad\quad\quad tmp \leftarrow \emptyset$;
12 $\quad\quad\quad$ **foreach** $\cdot \xrightarrow{\cdot \mid g^{=}, \cdot, up} \cdot \in T^{\bullet}$ **do**
13 $\quad\quad\quad\quad$ **if** $up(r_i) = y \neq \bot \land c(y) \neq 0$ **then** $tmp \leftarrow tmp \cup \{y\}$;
14 $\quad\quad\quad op_{r_i} \leftarrow tmp$;
15 $\quad\quad\quad c'(r_i) \leftarrow \sum_{x \in op_{r_i}}^{>1 \rightsquigarrow \omega} c(x)$;
16 $\quad\quad$ **foreach** $q' \in S'$ **do** `// check for Cartesian overapproximation`
17 $\quad\quad\quad P \leftarrow op_{r_1} \times \cdots \times op_{r_n}$ for $\{r_1, \ldots, r_n\} = \mathbf{R}[q']$;
18 $\quad\quad\quad$ **foreach** $(x_1, \ldots, x_n) \in P$ **do**
19 $\quad\quad\quad\quad$ **if** $\nexists (\cdot \xrightarrow{\cdot \mid \cdot, \cdot, up} q') \in T^{\bullet}$ *s.t.* $\bigwedge_{1 \leq i \leq n} up(r_i) = x_i$ **then return** $\bot$;
20 $\quad\quad up' \leftarrow \{r_i \mapsto op_{r_i} \mid r_i \in \mathbf{R}\}$;
21 $\quad\quad$ **if** $(S', c') \notin Q'$ **then**
22 $\quad\quad\quad \text{worklist}.push((S', c'))$;
23 $\quad\quad\quad Q' \leftarrow Q' \cup \{(S', c')\}$;
24 $\quad\quad \Delta' \leftarrow \Delta' \cup \left\{(S, c) \xrightarrow{a \mid g, \mathbf{R} \setminus g, up'} (S', c')\right\}$;
25 **return** $\mathcal{A}' = (Q', \mathbf{R}, \Delta', I', \{(S, c) \in Q' \mid S \cap F \neq \emptyset\})$;

Proof. Let us consider the language $L_{\neg\forall repeat}$ from the proof of Theorem 4.2, which is expressible using RSAs, and its complement $L_{\forall repeat}$, which is not expressible using RSAs. Since DRSAs are closed under complement (Theorem 4.4), if they could accept $L_{\neg\forall repeat}$, they could also accept $L_{\forall repeat}$, which is a contradiction. Therefore, $L_{\neg\forall repeat} \notin$ DRSA. □

4.2 Expressivity

The RSA model captures an interesting class of data word languages, strictly generalizing NRAs and being incomparable to ARAs [18] or pebble automata [48]. Due to the page limit, see [30] for detailed positioning of RSAs in the landscape of register automata models.

5 Determinizing Register Automata

RSAs have the following interesting property: a large class of NRAs can be determinized into DRSAs (we emphasize that the determinization considered here changes the model from one storing single values in registers (NRAs) to one storing sets of values in registers (DRSAs)). In this section, we give a determinization semi-algorithm and specify properties of a class of NRAs for which it is complete.

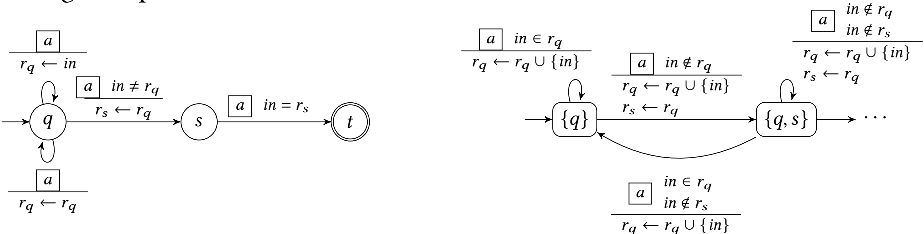
Let $\mathcal{A} = (Q, \mathbf{R}, \Delta, I, F)$ be an NRA. We use $\mathbf{R}[q]$ for $q \in Q$ to denote the set of registers r *active* at q, for which there exists a transition $s \xrightarrow{\cdot \mid g^{=}, g^{\neq}, up} t \in \Delta$ with (i) $up(r) \neq \bot$ and $t = q$ or (ii) $r \in g^{=} \cup g^{\neq}$ and $s = q$. Besides being the basis for the register locality optimization in Section 5.1.2, the set of active registers $\mathbf{R}[q]$ is also used in the basic algorithm as an overapproximation of the set of registers with a value different from $\bot$ (it excludes those register that were just assigned $\bot$). Given a set of states S, we define $\mathbf{R}[S] = \bigcup_{q \in S} \mathbf{R}[q]$. Furthermore, we call $\mathcal{A}$ *copyless* if there is no reachable configuration (q, f) such that $f(r_1) = f(r_2) \neq \bot$ for a pair of distinct registers $r_1, r_2 \in \mathbf{R}$, i.e., there is at most one copy of each data value in $\mathcal{A}$. Again, any NRA can be converted into the copyless form, however, the number of states can increase to $B_{|\mathbf{R}|} \cdot |Q|$ where B_n is the n-th Bell number. Intuitively, the transformation is done by creating one copy of each state for every possible partition of $\mathbf{R}$ (the partitions contain registers with the same value), and modifying the transition function correspondingly.

The determinization (semi-)algorithm for a copyless NRA $\mathcal{A}$ is shown in Algorithm 1. On the high level, it is similar to the classical Rabin-Scott subset construction for determinizing finite automata [52] with additional treatment of registers superimposed onto it.

During the construction, we track (i) all states of $\mathcal{A}$ in which the runs of $\mathcal{A}$ might be at a given point, represented by a set of states $S \subseteq Q$ and (ii) a mapping $c \colon \mathbf{R} \to \{0, 1, \omega\}$ assigning to each register its *size class*, that records whether the size of the set in the register is 0, 1, or larger than 1 (denoted by the ω). The size classes are needed to ensure that our simulation of a disequality test $in \neq r$ by the non-membership test $in \notin r$ is precise, as explained under the item (1) below. the macrostate is then a pair (S, c). The initial state of the constructed DRSA is the macrostate (I, c_0) where c_0 is a mapping assigning zero to each register (the run of a DRSA starts with all registers initialized to $\emptyset$) (Line 1).

The main loop of the algorithm then constructs successors of reachable macrostates for each $a \in \Sigma$ and each $g \subseteq \mathbf{R}$ on Line 5; each pair a, g corresponds to the so-called *minterm* (minterms denote combinations of guards whose semantics do not overlap [14]). The set of active registers $\mathbf{R}[S]$ is used here to prune those minterms that clearly cannot be satisfied, since they are testing a register whose value must be $\bot$. For each minterm, we collect all transitions of $\mathcal{A}$ compatible with this minterm (Line 6) and generate the successor set of states S' (Line 7). The $\mathcal{A}'$ update function up' for register r is then set to collect into r all possible values that might be stored into r in $\mathcal{A}$ on any run over the input word at the given position (Lines 10–20).
The algorithm uses the following three techniques to handle three sources of imprecision:

(1) *Register size classes. (Lines 10 to 15).* Since the algorithm collects in the set-register r all possible values that could have been stored into the standard register r in $\mathcal{A}$, if the disequality tests in $g^{\neq}$ were changed for non-membership tests in $g^{\notin}$, this could mean that $\mathcal{A}'$ might not be able to simulate some transition of $\mathcal{A}$ (the transition would not be enabled). Consider the following example:

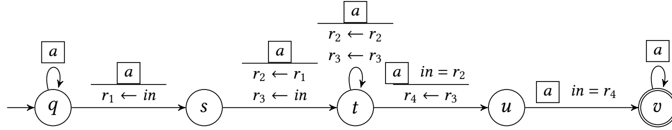


(a) An NRA $\mathcal{A}$ with a disequality guard (b) A part of the DRSA obtained for $\mathcal{A}$

where (b) contains a part of the DRSA obtained if Algorithm 1 did not use the c-component of macrostates. The reason for this is that after reading the third symbol (i.e., $\langle a, 2 \rangle$), the RSA

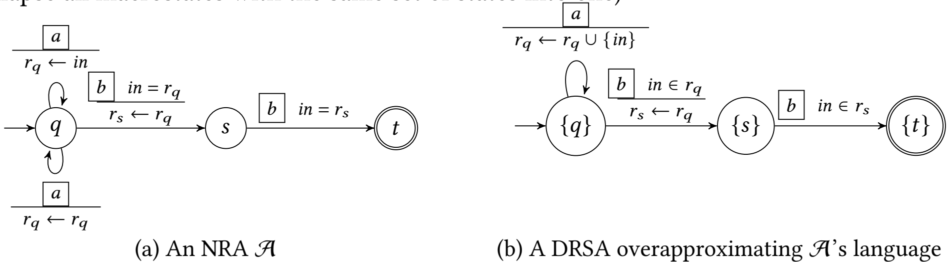
goes to the macrostate $\{q\}$—it thinks it cannot be in s any more. This is the reason why we augment macrostates with the c-component, the register size classes. They allow to detect that a disequality test is performed on a register containing more than one element, in which case we terminate the algorithm (Line 8). The sizes classes are updated on Line 15 where the sum is *saturated* to ω for values > 1 (denoted by $\overset{>1\rightsquigarrow\omega}{\sum}$).

(2) *Checking for Cartesian overapproximation (Lines 16 to 19).* By collecting all possible values that can occur in registers, the algorithm is performing the so-called *Cartesian overapproximation* (i.e., it is losing information about dependencies between components in tuples). This can lead to a scenario where, for some set-register assignment f' of $\mathcal{A}'$, we would have $d_1 \in f'(r_1)$ and $d_2 \in f'(r_2)$, but there would be no corresponding configuration of $\mathcal{A}$ with register assignment f such that $d_1 = f(r_1)$ and $d_2 = f(r_2)$. Consider, e.g., an NRA for the language $\{uvwvz \mid u, w, z \in (\Sigma \times \mathbb{D})^*, |v| = 2\}$:



When the algorithm computes the successor of the macrostate $(\{q, s, t\}, \{r_1{:}1, r_2{:}1, r_3{:}1\})$ over $a \in \Sigma$ and the guard $g = \emptyset$, it would obtain the following update of registers: $r_1 \leftarrow \{in\}$ (transition from q to s), $r_2 \leftarrow r_1 \cup r_2$ (transition from s to t and transition from t to t), and $r_3 \leftarrow r_3 \cup \{in\}$ (transition from s to t and transition from t to t). This would simulate also the update $r_2 \leftarrow r_2, r_3 \leftarrow in$, which is nowhere in the original NRA. The algorithm detects the possibility of such an overapproximation on Lines 16–19. Note that the overapproximation checking is overly conservative in a way that for some automata, if we did not do the check and continued the construction, outputting the DRSA, the resulting DRSA would still be correct (see [30]).[5]

(3) *Choice collapse at tests (Line 9).* When a set-register has collected several nondeterministic choices of values for a standard NRA register, then testing membership of a concrete value, which stands for testing equality of the NRA register, should collapse the choices to that single value. Without the collapse, a subsequent second membership test with a different value might pass as if the NRA register could hold two different values at once within a single non-deterministic case. Consider the following example of an NRA $\mathcal{A}$ and an RSA obtained from $\mathcal{A}$ by Algorithm 1 without the substitution $up[g^= \mapsto in]$ on Line 9 (to save space, we collapse all macrostates with the same set of states into one):



(a) An NRA $\mathcal{A}$ (b) A DRSA overapproximating $\mathcal{A}$'s language

One can see that while $\mathcal{A}$ cannot accept the word $\langle a, 1\rangle\langle a, 2\rangle\langle b, 1\rangle\langle b, 2\rangle$, the DRSA can. This happens because the DRSA did not "*collapse*" the possible nondeterministic choices that are kept in the registers for the value of r_q after the first membership test (on the transition

[5]In the implementation, we actually postpone the overapproximation test only after the whole DRSA is constructed. At this point, we can check more precisely whether there is some real overapproximation.

from $\{q\}$ to $\{s\}$) succeeded. We avoid this situation by the substitution on Line 9, which performs the collapse of the set of nondeterministic choices given by particular values of $g^=$ into a single value when it is positively tested. The update on the RSA transition from $\{q\}$ to $\{s\}$ constructed by the algorithm will then become $r_s \leftarrow \{in\}$ and the result will be precise. One might also imagine similar scenario as the previous but with several registers copying a nondeterministically chosen value (e.g., when a data value is copied from r_1 to r_2 and, later, r_1 is positively tested for equality, we need to guarantee that the value of r_2 also collapses to the given data value). In order to avoid this, we require that the input NRA is *copyless*, i.e., it never happens that a data value is in more than one register.

The correctness of Algorithm 1 is summarized by the following theorem, proved in [30].

Theorem 5.1. *When Algorithm 1 returns a* DRSA $\mathcal{A}'$, *then* $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}')$.

The following proposition establishes the numbers of states and transitions of the output DRSA of Algorithm 1, which follow directly from the structure of macrostates and transitions of $\mathcal{A}'$.

Proposition 5.2. *Let* $\mathcal{A} = (Q, \mathbf{R}, \Delta, I, F)$ *be an RA and let Algorithm 1 return a* DRSA $\mathcal{A}' = (Q', \mathbf{R}, \Delta', I', F')$. *Then* $|Q'| \leq 2^{|Q|+(\log_2 3)\cdot|\mathbf{R}|}$ *and* $|\Delta'| \leq |\Sigma| \cdot 2^{|Q|+(\log_2 6)\cdot|\mathbf{R}|}$.

We note that the lower bound on $|Q'|$ is $2^{|Q|}$, which comes from the lower bounds of determinization of finite automata [20, Section 1.4.1]. In the RAs obtained from regexes, typically $|Q| \gg |\mathbf{R}|$.

We can also modify Algorithm 1 to omit the c-component of macrostates (S, c), which will give us the bounds $|Q'| \leq 2^{|Q|}$ (i.e., the same as for finite automata determinization) and $|\Delta'| \leq |\Sigma| \cdot 2^{|Q|+|\mathbf{R}|}$. This simplification of the algorithm may in practice restrict the class of input RAs on which it successfully terminates. This is, however, not an issue for RAs *without* disequality guards $g^{\neq}$, which are output by our algorithm for converting regexes to RAs (cf. Section 6.1); for those, the modified algorithm successfully terminates in the same cases (the size of the result might, however, be different in both directions—on the one hand, having the c-component increases the maximum possible number of states, but on the other hand, keeping track of which registers are empty can avoid generation of transitions and states that can never be used in a run).

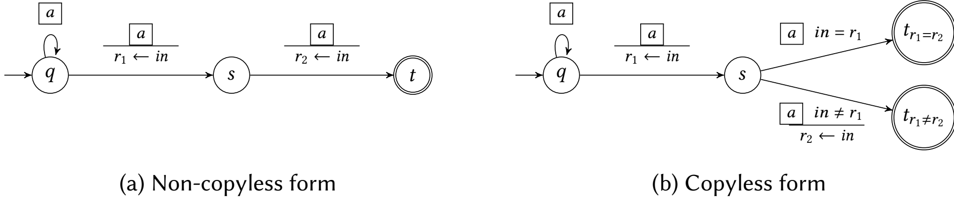
5.1 Improvements

In this section, we propose improvements of the determinization algorithm. In particular, we introduce slight modifications of Algorithm 1 as well as additional preprocessing of input NRAs aiming at enlarging the class of NRAs that can be determinized to DRSAs

5.1.1 Refining the Register Size Map. One approach to enlarge the class of NRAs that can be determinized by Algorithm 1 is to keep the value of c of each macrostate that overapproximate the register size as precise as possible. Indeed, the value of c affects the condition on Line 8. A way how c might become unnecessarily high is when $tmp \cap g \neq \emptyset$ and $in \in tmp$. In that case, $c(in) = 1$ does not need to be added to c' since according to the transition guard we know that in is in g. We hence modify the Line 14 to

$$op_{r_i} \leftarrow \begin{cases} tmp \setminus \{in\} & \text{if } tmp \cap g \neq \emptyset \text{ and} \\ tmp & \text{otherwise} \end{cases}$$

5.1.2 Register Locality. Another feature limiting the determinizability of the given NRA are nondeterministic transitions dealing with the same registers violating the Cartesian overapproximation checked on Line 19. In order to increase the locality of registers, we propose register-local NRAs. Locality of registers limiting updates of the same registers in different states and hence reducing the possibility of the condition on Line 19 holds true. Formally, we call $\mathcal{A}$ *register-local* if for all



(a) Non-copyless form

(b) Copyless form

Fig. 4. Copyless conversion for an NRA that stores data values in two registers.

$r \in \mathbf{R}$ it holds that if $r \in \mathbf{R}[q]$ and $r \in \mathbf{R}[s]$ for some states $q, s \in Q$, then $q = s$. It is easy to see that every NRA can be transformed into the register-local form by creating a new copy of a register for every state that uses it, potentially increasing the number of registers to $|Q| \cdot |\mathbf{R}|$. Formally, given a NRA $\mathcal{A} = (Q, \mathbf{R}, \Delta, I, F)$, we define its *register-localization* as $\mathcal{A}_\bullet = (Q, \mathbf{R}_\bullet, \Delta_\bullet, I, F)$ where $\mathbf{R}_\bullet = \{r_q \mid q \in Q, r \in \mathbf{R}\}$ and $\Delta_\bullet$ is defined as follows:

$$\Delta_\bullet = \Big\{ p \xrightarrow{a \mid g_\bullet^=, g_\bullet^{\neq}, up_\bullet} q \mid p \xrightarrow{a \mid g^=, g^{\neq}, up} q \in \Delta, \quad g_\bullet^= = v_p(g^=), \quad g_\bullet^{\neq} = v_p(g^{\neq}), \\ up_\bullet = \{(v_q(r), v_p(t)) \mid (r, t) \in up\} \cup \{(v_s(r), \bot) \mid s \in Q \setminus \{q\}, r \in \mathbf{R}\} \Big\} \tag{2}$$

where the localization function v_q is defined as $v_q(x) = x_q$ if $x \in \mathbf{R}$, otherwise $v_q(x) = x$ for $x \in \{in, \bot\}$. We extend the definition of the localization function to set of values in the usual way. As a prior step to Algorithm 1, we first convert input NRA to its register-local form.

5.1.3 Relaxing the Copyless Property. Conversion of an input NRA to an equivalent copyless NRA may introduce non-equality guards on registers having nondeterministically chosen values (i.e., in two runs over the same string, the registers can have different values at the same position in the string). Such non-equality guards can cause Algorithm 1 to return $\bot$ on Line 8 as it is shown in Example 5.3. In order to reduce the introduction of non-equality guards, we relax the copyless condition into *relation-free* condition in a way that there is no transition where multiple registers are assigned the same value regardless of the input. In other words, registers can hold the same value as long a run does not induce equality relations on the register values. Syntactically, relation-freeness can be expressed as each register must appear on a right-hand side of an update at most once, and at most one register can be updated by *in* or a register in the equality guard. Note that every NRA can be converted to a relation-free form, which is then used as an input of Algorithm 1.

Example 5.3. Consider the NRA over $\Sigma = \{a\}$ shown in Fig. 4a. It non-deterministically selects a data value to store in r_1 and then stores the following data value in r_2. Equivalent copyless NRA is shown in Fig. 4b. The NRA non-deterministically selects the data value for r_1 as well, but before storing the next data value in r_2, it must check that the data value is not already stored in r_1. If the data value is already stored in r_1, then it marks that $r_1 = r_2$ in its state control and does not actually store anything in r_2. With this construction, however, we necessarily introduce a non-equality guard on r_1, which eventually causes Algorithm 1 to return $\bot$ on Line 8.

5.2 Determinizability

Naturally, we wish to syntactically characterise the class of NRAs for which Algorithm 1 is complete. We observe that when we start with an $\mathrm{NRA}_1^=$ and apply the register localization the algorithm always returns a DRSA (the theorem is proved in [30]).

THEOREM 5.4. *For every* $\mathrm{NRA}_1^=$*, there exists a* DRSA *accepting the same language.*

Let $\mathcal{B}(\text{NRA}_1^=)$ be the class of languages that can be expressed using a Boolean combination of $\text{NRA}_1^=$ languages, i.e., it is the closure of $\text{NRA}_1^=$ languages under union, and intersection, and complement (it could also be denoted as $\mathcal{B}(\text{URA}_1^=)$).

Example 5.5. For instance, the language $L_{\exists,\neg\exists repeat}$ composed as the concatenation of $L_{\exists repeat}$ and $L_{\neg\exists repeat}$ with a delimiter, formally

$$L_{\exists,\neg\exists repeat} = L_{\exists repeat} \cdot \{\langle b, d\rangle \mid d \in \mathbb{D}\} \cdot L_{\neg\exists repeat},$$

is in $\mathcal{B}(\text{NRA}_1^=)$, since it is the intersection of languages

$$L_{\exists repeat} \cdot \{\langle b, d\rangle \mid d \in \mathbb{D}\} \cdot \{\langle a, d\rangle \mid d \in \mathbb{D}\}^* \text{ and } \{\langle a, d\rangle \mid d \in \mathbb{D}\}^* \cdot \{\langle b, d\rangle \mid d \in \mathbb{D}\} \cdot L_{\neg\exists repeat},$$

but is expressible neither by an NRA nor by a URA (URAs cannot express the part *before* the delimiter and NRAs cannot express the part *after* the delimiter). □

The determinizability of $\mathcal{B}(\text{NRA}_1^=)$ then follows directly from Theorems 4.4 and 5.4.

Corollary 5.6. *For any language in* $\mathcal{B}(\text{NRA}_1^=)$*, there exists a* DRSA *accepting it.*

A direct consequence of Corollary 5.6 and decidability of the emptiness check is also decidability of the inclusion problem as it is shown in the following corollary.

Corollary 5.7. *The inclusion problem between* RSA *and* $\mathcal{B}(\text{NRA}_1^=)$ *is decidable.*

Proof. We just write $\mathcal{L}(\mathcal{A}_1) \subseteq \mathcal{L}(\mathcal{A}_2)$ as $\mathcal{L}(\mathcal{A}_1) \cap \overline{\mathcal{L}(\mathcal{A}_2)} = \emptyset$ and use Corollary 5.6 and Theorems 4.1 and 4.2. □

6 Matching of Regexes with Backreferences

In this chapter, we define regexes with backreferences and show how they can be converted into NRAs using a construction based on Antimirov derivatives. We then establish the complexity of regex matching with DRSAs constructed from these regexes.

6.1 Regexes with Backreferences

Syntax. A *regex with backreferences* (REBR) $\mathcal{R}$ over alphabet Σ is defined inductively according to the following grammar:

$$\mathcal{R} ::= \epsilon \mid S \mid \mathcal{R} + \mathcal{R} \mid \mathcal{R} \cdot \mathcal{R} \mid \mathcal{R}^* \mid (S)_m \mid \backslash m$$

where $S \subseteq \Sigma$, and $m \in \mathbb{N}$. In addition to the standard regex syntax, the definition introduces (i) a *capture group* $(S)_m$ (explicitly indexed by m), capturing a single symbol from the set S, and (ii) a *backreference* $\backslash m$ to the m-th capture group. We use *RE* for the set of all REBRs. Note that compared to the languages that can be specified using, e.g., *Perl Compatible Regular Expressions* (PCREs) [32], our definition does not enable expressing languages where capture groups have unbounded length, e.g., the language defined by the PCRE `/^(.*)\1$/`, which matches strings of the form ww. Capture groups with bounded length (but longer than 1) can be expressed using REBRs by splitting the longer capture groups into single-letter ones, e.g., PCRE `/(...)\1/` can be expressed as `/(.)(.)(.)\1\2\3/`.

Semantics. We will define the semantics of REBR $\mathcal{R}$ in two phases: 1) in terms of *annotated words*, with marked parts matched by capture-groups and "pointers" to capture groups in place of backreferences; and 2) normal words obtained by interpreting the annotations.

In phase 1, subwords matching capture groups are annotated by the group's index m and the keyword in, and backreference annotations consist of the index of a capture group and the keyword ref. The language of the annotated words $\mathcal{L}_{\mathtt{ref}}$ is hence a language over the alphabet $\Sigma_{\mathtt{ref}} = \Sigma \cup \{\langle s, m, \mathtt{in}\rangle, \langle m, \mathtt{ref}\rangle \mid s \in \Sigma, 0 \leq m \leq k_{\mathcal{R}}\}$, with $k_{\mathcal{R}}$ being the maximum index of a capture group or a backreference in $\mathcal{R}$, defined as follows:

$$\mathcal{L}_{\mathtt{ref}}(\epsilon) = \{\epsilon\} \qquad \mathcal{L}_{\mathtt{ref}}(\mathcal{R}_1^*) = (\mathcal{L}_{\mathtt{ref}}(\mathcal{R}_1))^*$$
$$\mathcal{L}_{\mathtt{ref}}(S) = S \qquad \mathcal{L}_{\mathtt{ref}}((S)_m) = \{\langle s, m, \mathtt{in}\rangle \mid s \in S\}$$
$$\mathcal{L}_{\mathtt{ref}}(\mathcal{R}_1 + \mathcal{R}_2) = \mathcal{L}_{\mathtt{ref}}(\mathcal{R}_1) \cup \mathcal{L}_{\mathtt{ref}}(\mathcal{R}_2) \qquad \mathcal{L}_{\mathtt{ref}}(\backslash m) = \{\langle m, \mathtt{ref}\rangle\}$$
$$\mathcal{L}_{\mathtt{ref}}(\mathcal{R}_1 \cdot \mathcal{R}_2) = \mathcal{L}_{\mathtt{ref}}(\mathcal{R}_1) \cdot \mathcal{L}_{\mathtt{ref}}(\mathcal{R}_2)$$

Intuitively, a backreference should match the character that the corresponding capture group matched last in the prefix left of the backreference. Formally, for the m-th capture group and a prefix w, it is the character $\mathtt{match}(w, m) = c$ if it is possible to write w as $w_p \cdot \langle c, m, \mathtt{in}\rangle \cdot w_s$ where $w_p \in \Sigma^*_{\mathtt{ref}}$ and $w_s \in \Sigma^*_{\mathtt{ref}}$ does not contain *any occurrence* of a symbol $\langle a, m, \mathtt{in}\rangle$ for arbitrary $a \in \Sigma$ (hence this split of the prefix w indeed marks the last match of the capture group in it). Otherwise, if w has no occurrence of an annotated capture group match $\langle a, m, \mathtt{in}\rangle$, then $\mathtt{match}(w, m) = \bot$.

In order to obtain a language over the alphabet Σ from $\mathcal{L}_{\mathtt{ref}}$, we define *annotation interpretation*, denoted as interp, which replaces the backreference annotations with the captured characters and removes the capture group annotations. Formally, given a word $w \in \Sigma^*_{\mathtt{ref}}$ and an index $1 \leq i \leq |w|$, interp is defined as

$$\mathtt{interp}(w, i) = \begin{cases} \mathtt{match}(w[1:i-1], m) & \text{if } w[i] = \langle m, \mathtt{ref}\rangle, \\ c & \text{if } w[i] = \langle c, m, \mathtt{in}\rangle, \\ w[i] & \text{otherwise,} \end{cases}$$

where $w[1:i] = w_1 \dots w_i$ for $w = w_1 \dots w_{|w|}$. We lift the definition to a word as $\mathtt{interp}(w) = \{w' \mid w' = \mathtt{interp}(w, 1) \cdots \mathtt{interp}(w, |w|), w' \text{ does not contain } \bot\}$. Notice that removing the words containing $\bot$ removes exactly cases where a backreference occurs before the corresponding capture group is matched. The *language* of a REBR $\mathcal{R}$ is then defined as $\mathcal{L}(\mathcal{R}) = \bigcup_{w \in \mathcal{L}_{\mathtt{ref}}(\mathcal{R})} \mathtt{interp}(w)$. Since our automaton model works with data words and the semantics of REBRs are defined as languages over Σ, we define the extension of words/languages over Σ to data words. For that we consider some *fixed* linear ordering on Σ, meaning that there is an injective mapping $\mathrm{ord}\colon \Sigma \to \mathbb{N}$. Then, for a word $w = a_1 \cdots a_n \in \Sigma^*$, we define $w^\bullet = \langle a_1, \mathrm{ord}(a_1)\rangle \cdots \langle a_n, \mathrm{ord}(a_n)\rangle$. We lift the definition to languages $\mathcal{L}^\bullet$ in the usual way.

Antimirov Derivatives. We can translate REBRs to RAs using a construction based on *Antimirov derivatives* [3]. A *partial derivative* is a function $\partial\colon \Sigma \times RE \to 2^{RE \times \mathbb{N}_\bot \times \mathbb{N}_\bot}$ (we use $\mathbb{N}_\bot$ to denote the set $\mathbb{N} \cup \{\bot\}$) such that $\partial_a(\mathcal{R})$ gives us all possible REBRs (together with potential information about writing or testing a register) obtained from $\mathcal{R}$ by trying to match a string with leading symbol $a \in \Sigma$. The definition of the derivatives is in Fig. 5. In the definition, we also use the predicate *nullable*($\mathcal{R}$), which denotes that $\mathcal{R}$ is *nullable*, i.e., it accepts the empty string. We note that our class of REBRs allows this relatively simple extension of Antimirov derivatives; if one considered unbounded capture groups, the derivatives would get more complex.

$$\partial_a(\epsilon) = \emptyset \qquad nullable(\epsilon) \Leftrightarrow true$$

$$\partial_a(S) = \begin{cases} \{\langle \epsilon, \bot, \bot \rangle\} & \text{if } a \in S \\ \emptyset & \text{otherwise} \end{cases} \qquad nullable(S) \Leftrightarrow false$$

$$\partial_a(r_1 + r_2) = \partial_a(r_1) \cup \partial_a(r_2) \qquad nullable(r_1 + r_2) \Leftrightarrow nullable(r_1) \vee nullable(r_2)$$

$$\partial_a(r_1 \cdot r_2) = \begin{cases} \alpha_a(r_1 \cdot r_2) \cup \partial_a(r_2) & \text{if } nullable(r_1) \\ \alpha_a(r_1 \cdot r_2) & \text{otherwise} \end{cases} \qquad nullable(r_1 \cdot r_2) \Leftrightarrow nullable(r_1) \wedge nullable(r_2)$$

$$\text{where } \alpha_a(r_1 \cdot r_2) = \{\langle r \cdot r_2, tst, up \rangle \mid \langle r, tst, up \rangle \in \partial_a(r_1)\}$$

$$\partial_a(r^*) = \{\langle r' \cdot r^*, tst, up \rangle \mid \langle r', tst, up \rangle \in \partial_a(r)\} \qquad nullable(r^*) \Leftrightarrow true$$

$$\partial_a((S)_m) = \{\langle r, \bot, r_m \rangle \mid \langle r, \bot, \bot \rangle \in \partial_a(S)\} \qquad nullable((S)_m) \Leftrightarrow false$$

$$\partial_a(\backslash m) = \{\langle \epsilon, r_m, \bot \rangle\} \qquad nullable(\backslash m) \Leftrightarrow false$$

Fig. 5. Antimirov derivatives for REBRs. The predicate $nullable(\mathcal{R})$ outputs *true* iff $\epsilon \in \mathcal{L}^\bullet(\mathcal{R})$.

NRA Construction. We use the partial derivatives defined above to transform a regex $\mathcal{R}$ into an NRA in the function rebr2ra. Let $k_\mathcal{R}$ be the maximum index of a capture group or a backreference occurring in $\mathcal{R}$. Then we define the NRA $\text{rebr2ra}(\mathcal{R}) = (Q, \{r_0, \ldots, r_{k_\mathcal{R}}\}, \Delta, \{\mathcal{R}\}, F)$ where

- the set of states Q contains all regexes that are reachable from $\mathcal{R}$ by the partial derivative construction, i.e., $Q = \mu Z\colon \bigcup\{q' \mid \langle q', \cdot, \cdot \rangle \in \partial_a(q), q \in Z, a \in \Sigma\} \cup \{\mathcal{R}\} \cup Z$,
- Δ is the smallest set such that if $\langle q', t, u \rangle \in \partial_a(q)$ for $a \in \Sigma$, then $q \xrightarrow{a \mid g^=, \emptyset, up} q' \in \Delta$ where
  - $g^= = \{r_i\}$ if $u = i \neq \bot$, else $g^= = \emptyset$ and
  - $up = \{r_i \mapsto r_i \mid 0 \leq i \leq k_\mathcal{R}, i \neq j\} \cup \{r_j \mapsto in\}$ if $u = j \neq \bot$, else $up = \{r_i \mapsto r_i \mid 0 \leq i \leq k_\mathcal{R}\}$, and
- $F = \{q \in Q \mid nullable(q)\}$.

Note that the number of used registers matches the number of capture groups/backreferences in $\mathcal{R}$. The argument that the number of states of $\text{rebr2ra}(\mathcal{R})$ is finite is similar as the one in [3]; more precisely, the number of states of the output NRA is in $O(|\mathcal{R}|)$. Then, using the properties of Antimirov derivatives and the construction described above, we have the following theorem.

THEOREM 6.1. *For a regex $\mathcal{R}$ we have $\mathcal{L}(\text{rebr2ra}(\mathcal{R})) = [\![\mathcal{R}]\!]^\bullet$.*

6.2 Regex Matching

In order to match a word $w \in \Sigma^*$ w.r.t. a given REBR $\mathcal{R}$, we first construct an RA $\mathcal{A}$ corresponding to $\mathcal{R}$ (using rebr2ra). Then, after register localization, we use Algorithm 1 to determinize $\mathcal{A}$.[6] If the algorithm does not return $\bot$, we have a DRSA $\mathcal{A}'$ representing $\mathcal{R}$. When testing regex membership, we check that $w^\bullet \in \mathcal{L}(\mathcal{A}')$ using an algorithm tracking configurations obtained during reading of $w^\bullet$. The following lemma establishing the complexity of this algorithm.

LEMMA 6.2. *Let w be a data word and $\mathcal{A} = (Q, \mathbf{R}, \Delta, I, F)$ be a DRSA over a finite data domain $\mathbb{D}$. Checking that $w \in \mathcal{L}(\mathcal{A})$ can be done in time $O(|\mathbb{D}| \cdot |w| \cdot |\mathbf{R}|^2)$.*

PROOF SKETCH. Consider a data word w and a DRSA $\mathcal{A} = (Q, \mathbf{R}, \Delta, I, F)$. Further let $\mathcal{D} = \{v_i \mid v_1 \ldots v_n = \mathbb{D}[w]\}$ (note that $|\mathcal{D}| \leq |\mathbb{D}|$). In this proof and the following text we assume a unit cost

[6] Regarding the determinizability of RAs obtained from REBRs, from Theorem 5.4 and the properties of the RA construction we have that we are able to process arbitrary REBRs containing a single capture group (but possibly multiple backreferences).

of comparison between elements of $\mathbb{D}$. The membership checking algorithm tracks configurations of the form (q, r) where $q \in Q$ and $r\colon \mathbf{R} \to 2^{\mathcal{D}}$. We assume that set-register values are stored in a sparse-set data structure with constant insertion and membership and with the union of two sets linear to the size of one of the two sets (elements of one sets are inserted to the other, creation of a new sparse-set is not needed as the number of registers is fixed). When constructing a next-state configuration, we first need to check transition guards to get a successor transition. Here, we assume that the successor transitions are kept in a BDD-like structure. Hence, for each register, it suffices to check whether the input symbol is in the register, and then use this bitvector to find an appropriate transition (recall the automaton is deterministic). This can be done in $O(|\mathbf{R}|)$. Then, we need to compute the transition update, which can be done in $O(|\mathbb{D}| \cdot |\mathbf{R}|^2)$. Therefore the overall complexity is $O(|\mathbb{D}| \cdot |w| \cdot |\mathbf{R}|^2)$. □

Note that if the data domain is infinite, testing $w \in \mathcal{L}(\mathcal{A})$ can be done in time $O(|w|^2 \cdot |\mathbf{R}|^2)$. Directly from Lemma 6.2 we have the following theorem.

Theorem 6.3. *For a fixed REBR $\mathcal{R}$ over a fixed alphabet, the worst-case time complexity of* DRSA-*based regex matching is* linear *to the length of the input word (provided Algorithm 1 finishes on the RA obtained from $\mathcal{R}$).*

6.3 Finer Complexity Analysis

The constant character cost complexity of matching in Theorem 6.3 hides a quadratic dependency on the number of registers $|\mathbf{R}|$ and the linear dependency on the size of the data domain $|\mathbb{D}|$. When the DRSA is obtained from a REBR of the size m and with r backreferences by (i) our derivative-based construction, (ii) register localization, and (iii) our determinization (Algorithm 1), then we have $|\mathbf{R}| \in O(m \cdot r)$. The $|\mathbb{D}|$ particularly can be large in practice and problematic.

We will argue that the factor $|\mathbb{D}|$, coming from the need to unite and copy registers, can be avoided if the automaton never copies registers. Copying is thus never done and the cost of uniting registers is covered by the maximum cost of adding elements to the registers. The quadratic dependence on $|\mathbf{R}| = rm$ also decreases to linear. Formally, we define an RSA as *copy-free* if it has no transition where it assigns the same non-singleton register to two registers. Practical relevance of this class is witnessed by the fact that in our experiments, the copy-free DRSA were constructed in 488 cases out of 1,335 (our main benchmark set of single-letter regexes that were found ReDoS prone).

Theorem 6.4. *For a copy-free DRSA, the complexity of testing membership is $O(|\mathbf{R}| \cdot |w|)$.*

Proof Sketch. Let us measure the age of a data value by when it was read form the input, and let the age of a register be the age of its oldest value. Let us union two registers by inserting values from the younger one to the older register. With sparse-sets, the complexity would be linear to the size of the younger register, as every insertion is constant-time. That is, within one union operation, each element in the younger register contributes $O(1)$. Note that since the automaton is copy-free, a value taken from a position in the input data word can be this way (within the union operation) inserted into older registers at most $|\mathbf{R}|$ times (after $|\mathbf{R}|$ unions, the value must be in the oldest register). Hence, inserting the value from a particular position in the input word within unions may take at most $O(|\mathbf{R}|)$ time, and we have $|w|$ positions, which amounts to $O(|\mathbf{R}| \cdot |w|)$ for uniting registers overall. Initialising the sparse-sets would take $O(|\mathbf{R}| \cdot |\mathbb{D}|)$ time, but $|\mathbb{D}|$ can be w.l.o.g. capped at $|w|$. □

Note that also testing $w \in \mathcal{L}(\mathcal{A})$ with an infinite $\mathbb{D}$ with copy-free DRSA would be in $O(|\mathbf{R}| \cdot |w|)$.

7 Experimental Evaluation

We have implemented a prototype RSA-based regex matcher rsamatch [72] in Python and evaluated its performance against other state-of-the-art regex matchers on realistic ReDoS attack vectors involving backreferences. Our experiments are designed to evaluate the following hypotheses:

(1) Existing regexes matchers are vulnerable when matching regexes with backreferences.
(2) A large portion of regexes with backreferences contain only single-letter backreferences.
(3) Most of the regexes with single-letter backreferences that are vulnerable to ReDoS can be converted to DRSAs and matched with a predictable performance.

7.1 Experimental Setup

Regex Matchers and Data Sets. We compared rsamatch against a representative sample of state-of-the-art matchers, namely, grep [21], pcre2 [32], Python RE [41] (denoted as re), standard library matchers of JavaScript [19], Java [50], and .NET [44] (denoted as js, java, and .net respectively). These are mostly backtracking-based, possibly in combination with other techniques, but they all mainly use their backtracking core when confronted with backreferences. We note that matchers for some scripting languages (Python's re and pcre2) are actually implemented in C and not in the scripting language itself. Some very fast matchers such as RE2 [28] and especially HyperScan [73], based on automata and other techniques, are missing in the comparison since they do not support backreferences [27, 34]. All experiments were run on a Ubuntu GNU/Linux machine with the AMD EPYC 9124 CPU with 16 cores at 3.0 GHz and 94 GiB of memory (we note that the memory consumption of all matchers was far below the limits imposed by the hardware).

As our regexes, we started with 3,252 real-world regexes with backreferences obtained from the comprehensive data set Lingua Franca [17] and 8,794 regexes from the benchmarks of the ReDoS generator Rengar [74], for a total of 12,046 regexes.[7] From these, we picked regexes with single-letter backreferences because although multi-letter capture groups can be rewritten as single-letter ones (for a bounded length, cf. Section 6), the determinization often fails for them (Algorithm 1 returns $\perp$ due to overapproximation), which gave us 3,299 regexes (27 %).

ReDoS Inputs. In general, there are several flavours of ReDoS considered in the literature. In our evaluation, we consider the scenario where there is a regex (known to the attacker) running on a server (or a network intrusion detection system or a packet filter etc.) and the attacker provides an input that exhibits a costly matching, which is, to the best of our knowledge, the most realistic setting, since it has already led to several real-world ReDoSes [4, 22].[8] The input of every experiment is a pair of a regex and an input string, which was created using the ReDoS generator Rengar [74]. Rengar was able to provide an attack vector for 1,335 regexes from the data set (40 %). Since in the considered ReDoS setting, one regex is used to match a large quantity of user inputs, we perform the determinization in rsamatch offline and then run the matching with the precomputed DRSA. Out of the 1,335 regexes with ReDoS inputs, our determinization algorithm returned $\perp$ on 43 regexes (3 %) and timed out (because of a regex complexity other than backreferences; e.g., most of the regexes start with an implicit `".*"`, which already creates a nondeterministic choice at the beginning) on 41 of them (3 %). The average runtime of the determinization algorithm on successful

[7] In the Lingua Franca data set, 52 % of the regexes had capture groups of fixed lengths and 57 % had capture groups of bounded lengths (the fixed-length are a subset of the bounded-length). In the Rengar data set, 22 % regexes had capture groups of fixed lengths and 24 % had capture groups of bounded lengths. In total, 30 % regexes had capture groups of fixed lengths and 33 % had capture groups of bounded lengths.

[8] Some of the other settings considered in the literature is, e.g., when the attacker can provide both the regex and the input. While such a situation may occur in the real-world, it is less common and much easier to exploit when backreferences are enabled due to the NP-hardness of their matching [2].

Table 1. A frequency table of run times and statistics for rsamatch and the other matchers on the 1,246 supported regexes with the Rengar-generated inputs. The column **TOs** denotes timeout (> 100 s), column **> 1 s** is the sum of the values in all columns over 1 s, and the **errors** column shows the numbers of unsupported regexes. The numbers in the right-hand part of the table are for successful runs only and are in seconds.

tool	< 1 s	1–5 s	5–10 s	10–50 s	50–100 s	TOs	> 1 s	errors	mean	median	std. dev
rsamatch	1,244	2	0	0	0	0	2	—	0.299	0.16	0.303
grep	1,131	6	0	16	4	15	41	74	0.664	0.01	5.973
re	1,139	36	14	11	3	43	107	0	0.532	0.01	4.416
pcre2	1,001	14	4	4	0	3	42	203	0.144	0.01	1.219
js	660	8	1	11	1	33	54	532	0.531	0.06	3.702
java	595	18	5	7	5	19	54	597	0.996	0.05	6.438
.net	629	24	13	9	3	40	89	528	0.813	0.04	5.165

instances was 1.150 s, the median was 0.260 s, and the standard deviation was 3.064 s. Since the determinization needs to be performed only once for many runs of the matching algorithm, the cost of determinization can be amortized. Rengar generated strings of the length between 55 and 90,124 characters, with the average of 34,521 characters. We set the timeout to quite generous 100 s.

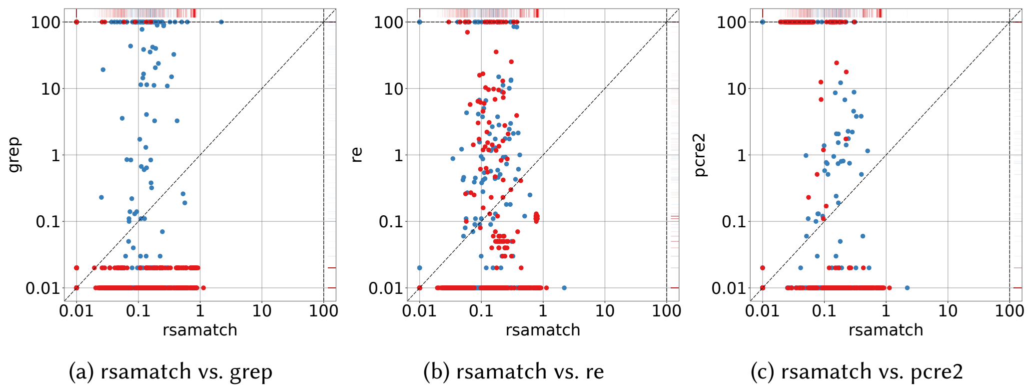
Determinizability. In the previous paragraph, we analyzed on how many of the ReDoS-prone regexes we managed to construct a corresponding DRSA. Let us now discuss on how many of all input regexes (12,046) we managed to build the corresponding DRSA. From the 12,046 regexes, 2,944 contained unsupported syntax (we use Python frontend and its regex dialect), which gave us 9,102 regexes to start with. Out of these, our algorithm outputs DRSA on 3,138 of them (34.5 %). When we restrict ourselves to the single-letter ones, these were 3,434, making our algorithm successful on 91 % of them.

7.2 Results

In Table 1, we show a frequency table (the table form of a histogram) of run times of rsamatch and the other matchers on the 1,246 supported input regexes. We note that some of the regex matchers failed on a considerable number of the input regexes—this is due to different dialects of regexes used by the matchers and should not be considered their failure (rsamatch uses Python's dialect, which is why re has 0 errors). The use of different dialects means that the semantics of one regex differs based on the matcher [17]; while this would be an issue if we were are cross-comparing the performance of the matchers, in this study, we are focusing on ReDoS vulnerability, where the concrete semantics of a regex is not so important.

The results show that rsamatch's run time for the vast majority of benchmarks (all except two) is below one second (the run times for the two hardest benchmarks were 1.13 and 2.2 seconds). The other matchers, on the other hand, indeed struggled on these attack vectors considerably, often taking tens of seconds or even exceeding the 100 s timeout. Considering that the normal matching time on such strings is counted in milliseconds, they can indeed be considered manifestations of vulnerabilities, even for the relatively simple class of regexes with single-letter backreferences. If we fix a ReDoS threshold at 1 s, the column **> 1 s** counts the number of successful ReDoS attacks. Here, the second best matcher (after rsamatch) was pcre2 with 42 attacks (although it could not run on 203 benchmarks) and then grep with 41 attacks (with 74 benchmarks excluded due to an error).

In the right-hand part of the table, we give statistics about the run times of the matchers on the instances where they did not timeout (or fail): the arithmetic mean, the median, and the standard deviation. Although rsamatch is a prototype written in Python, its average time is comparable to the other matchers (also taking into account the fact that they timed out on a number of inputs). The median run time of rsamatch is higher than for the other matchers, but it should be significantly



(a) rsamatch vs. grep (b) rsamatch vs. re (c) rsamatch vs. pcre2

Fig. 6. Comparison of rsamatch with grep, re, and pcre2. Times are in seconds, axes are logarithmic. Dashed lines represent timeouts (100 s). Colours distinguish datasets: • Lingua Franca, • RENGAR.

decreased if reimplemented in a more efficient programming language, such as C/C++. The last column of the table also exhibits the robustness and predictability of the performance of rsamatch, since the standard deviation of the run times is (despite rsamatch's prototype nature) an order of magnitude lower than for the other matchers.

We include a more detailed comparison with the best competing tools, grep, re, and pcre2, in the form of scatter plots shown in Fig. 6. One can clearly see the contrast in performance predictability, with our matcher almost always staying below the 1 s mark, on many being instances significantly faster than the competitors. Note that grep either managed to solve the matching problem almost instantly or timed out.

8 Related Work

Regex Matching. The literature on efficiently matching regexes is rich, with works ranging from matching feature-rich regexes in mainstream languages [7, 71], to instruction-level optimized regex matching in HyperScan [73], down to accelerating regex matching using FPGAs [11, 42]. Here we focus mainly on automata-based regex matching and regexes with backreferences.

Automata-based regex matching can be traced back to Thompson [66], who introduced a practical algorithm based on on-demand NFA determinization with caching. There have been several attempts of automata models for matching regexes with backreferences, but they have generally failed expectations of the community. Memory automata of Schmid [56] provide a natural extension of finite automata with a register bank, each register able to store a word, but do not allow determinization. The language-theoretic aspects of regexes with backreferences were further studied in [25]. Chida and Terauchi [13] show that lookaheads increase the expressive power of regexes with backreferences. Becchi and Crowley [8] encode backreferences into an extended model of NFAs using backtracking. Nogami and Terauchi [49] devise a quadratic-time (w.r.t. the input length) algorithm for matching a certain class of regexes with backreferences, which is incomparable to our class—they allow capture groups with unbounded length, but only one capture group and one reference to it; our work, on the other hand, allows multiple capture groups, multiple references to them, and our constructed automata often process the input in linear time (with a quadratic time worst-case guarantee). Namjoshi and Narlikar [47] extend Thompson's algorithm [66] to a model similar to the models of [8, 56] in the obvious way by tracking the set of all possible configurations, which is inefficient (for each symbol, the number of operations proportional to the (potentially exponential) size of the set of configurations is required). Varatalu *et al.* [71] present

a derivative-based regex matcher supporting many extensions of classical regexes. They do not support backreferences, but it seems that our derivatives (cf. Section 6) could be combined with it to extend its usability. Moreover, there are also some proprietary *ad hoc* solutions that allow matching of regexes with backreferences [64], but they are mostly incomplete and with no guarantees.

In our work, we were inspired by *counting-set automata* introduced by Turoňová *et al.* [69], which use *sets of counter values* to compactly represent configurations of *counting automata* [33] (a restricted version of counter automata [45] with a bound on the value of counters for compact representation of finite automata), to obtain a deterministic model for efficient matching of regular expressions with repetitions.

ReDoS. While initially neglected, the potential of exploiting regex matching for denial of service was popularized by the successful attacks on StackOverflow [22] and `Express.js` [4]. While many industrial matchers handle basic regexes reasonably well [7, 21, 28, 71, 73], using extensions can make the matchers vulnerable to ReDoS; in [68], this was demonstrated even for highly optimized matchers such as RE2 [28] or HyperScan [73] (these two have backreferences explicitly disabled [27, 34]). Over the years, there have been many works dealing with the analysis of regexes for ReDoS vulnerability, sometimes including ReDoS generation, and proposals for how to avoid this attack vector (see, e.g., the works in [6, 16, 17, 29, 38, 40, 43, 51, 62, 63, 74–76] or the systematic literature review in [9]). On a more theoretical level, Terauchi [65] studies conditions for ReDoS vulnerability of regexes with backreferences and proposes a technique for transforming a class of memory automata to invulnerable regexes.

Automata Models for Data Words. The literature on automata over infinite alphabets is rich, see, e.g., the excellent survey by Segoufin [61] and the paper by Neven *et al.* [48]. Register automata were introduced (under the name *finite memory automata*) by Kaminski and Francez in [37] and their basic properties further studied by Sakamoto and Ikeda in [55]. Demri and Lazić study in [18] (non-)deterministic, universal, and alternating one-way and two-way register automata, and their relation to the linear temporal logic with the *freeze* quantifier, which can store the current data value into a register. In particular, they show that $\mathrm{LTL}_1^{\downarrow}(\mathsf{X}, \mathsf{U})$, i.e., the fragment of the logic with one freeze register and the *next* (X) and *until* (U) temporal operators captures the class of languages accepted by one-way alternating register automata with one register (ARA_1). Figueira [23] introduces alternating register automata with one register and two extra operations: guess and spread ($\mathrm{ARA}_1(\mathrm{g}, \mathrm{s})$). Intuitively, guess and spread can be used for existential and universal quantification over future and past data values respectively.

Set-augmented finite automata, introduced in [5], resemble RSAs in a way that they allow to store a set of values in a register, however, with noticeable restrictions: (i) set-augmented automata allow equality/disequality guard to be a singleton only, and (ii) the update function is restricted to storing the currently read data value to a single register (or no update is applied as a second option). It is not possible, e.g., to union the contents of two set-registers (which is a crucial operation in our determinization algorithm), or to empty a register. Due to these restrictions, RSAs have greater expression power compared to set-augmented automata (e.g., the parametric language defined in [5, Theorem 1] can be accepted by RSA_1 but not by a corresponding set-augmented automaton).

History-register automata, introduced in [70], are a similar model to RSA^{rm} in a way that they also allow sets of values to be stored in registers. The key difference is the assignment of values to registers. The only way how to change a value in a register is add/remove the *currently processed* data value. It is not possible e.g., to union the content of two registers (which is, as noted above, crucial in our determinization), which is possible in RSA^{rm} making our model more expressive. In particular,

in [30] we show that RSA^{rm} are expressive at least as history-register automata and moreover, deterministic history-register automata are strictly less expressive than deterministic RSA^{rm}.

9 Conclusion and Future Work

We have introduced register set automata, a class of automata over data words providing an underlying formal model for efficient matching of a subclass of regexes with backreferences. There are many challenges that we wish to address in the future: (i) improvement of our determinization algorithm to work on a larger class of input NRAs, (ii) explore other, more expressive, formal models that would allow deterministic automata models for a large class of regexes with backreferences (e.g., the hard regex in footnote[2] is beyond the power of DRSAs), (iii) develop efficient toolbox for working with RSAs occurring in practice, and (iv) explore other approaches of how to efficiently deal with nondeterminism in practice, e.g., in a similar way as considered in [39] (which does not consider backreferences).

Data Availability Statement

An environment with the tools and data used for the experimental evaluation in the current study is available at [31].

Acknowledgments

We thank the anonymous reviewers for their feedback that improved the quality of the paper. This work was supported by the Czech Science Foundation projects 26-22640S and 25-17934S and the FIT BUT internal project FIT-S-26-9011. This work has been executed under the project VASSAL: "Verification and Analysis for Safety and Security of Applications in Life" funded by the European Union under Horizon Europe WIDERA Coordination and Support Action/Grant Agreement No. 101160022.

References

[1] Adar Weidman. 2025. Regular expression Denial of Service — ReDoS. https://owasp.org/www-community/attacks/Regular_expression_Denial_of_Service_-_ReDoS. [Online; accessed 25-March-2025].

[2] Alfred V. Aho. 1990. Algorithms for Finding Patterns in Strings. In *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity*, Jan van Leeuwen (Ed.). Elsevier and MIT Press, 255–300.

[3] Valentin M. Antimirov. 1996. Partial Derivatives of Regular Expressions and Finite Automaton Constructions. *Theor. Comput. Sci.* 155, 2 (1996), 291–319. doi:10.1016/0304-3975(95)00182-4

[4] Adam Baldwin. 2016. Regular Expression Denial of Service affecting Express.js. https://medium.com/node-security/regular-expression-denial-of-service-affecting-express-js-9c397c164c43. [Online; accessed 25-March-2025].

[5] Ansuman Banerjee, Kingshuk Chatterjee, and Shibashis Guha. 2023. Set Augmented Finite Automata over Infinite Alphabets. In *Developments in Language Theory*. Springer Nature Switzerland, Cham, 36–50.

[6] Efe Barlas, Xin Du, and James C. Davis. 2022. Exploiting Input Sanitization for Regex Denial of Service. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 883–895. doi:10.1145/3510003.3510047

[7] Aurèle Barrière and Clément Pit-Claudel. 2024. Linear Matching of JavaScript Regular Expressions. *Proc. ACM Program. Lang.* 8, PLDI (2024), 1336–1360. doi:10.1145/3656431

[8] Michela Becchi and Patrick Crowley. 2008. Extending finite automata to efficiently match Perl-compatible regular expressions. In *Proceedings of the 2008 ACM CoNEXT Conference on - CONEXT '08*. ACM Press, Madrid, Spain, 1–12. doi:10.1145/1544012.1544037

[9] Masudul Hasan Masud Bhuiyan, Berk Çakar, Ethan H. Burmane, James C. Davis, and Cristian-Alexandru Staicu. 2025. SoK: A Literature and Engineering Review of Regular Expression Denial of Service (ReDoS). In *Proceedings of the 20th ACM Asia Conference on Computer and Communications Security, ASIA CCS 2025, Hanoi, Vietnam, August 25-29, 2025*. ACM, 1659–1675. doi:10.1145/3708821.3733912

[10] Benedikt Bollig, Peter Habermehl, Martin Leucker, and Benjamin Monmege. 2013. A Fresh Approach to Learning Register Automata. In *Developments in Language Theory (Lecture Notes in Computer Science)*. Springer, Berlin, Heidelberg, 118–130. doi:10.1007/978-3-642-38771-5_12

[11] Milan Ceska, Vojtech Havlena, Lukás Holík, Jan Korenek, Ondrej Lengál, Denis Matousek, Jirí Matousek, Jakub Semric, and Tomás Vojnar. 2019. Deep Packet Inspection in FPGAs via Approximate Nondeterministic Automata. In *27th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM 2019, San Diego, CA, USA, April 28 - May 1, 2019*. IEEE, 109–117. doi:10.1109/FCCM.2019.00025

[12] Yu-Fang Chen, Vojtěch Havlena, Ondřej Lengál, and Andrea Turrini. 2020. A Symbolic Algorithm for the Case-Split Rule in String Constraint Solving. In *Programming Languages and Systems*, Vol. 12470. Springer International Publishing, Cham, 343–363. doi:10.1007/978-3-030-64437-6_18 Series Title: Lecture Notes in Computer Science.

[13] Nariyoshi Chida and Tachio Terauchi. 2023. On Lookaheads in Regular Expressions with Backreferences. *IEICE Trans. Inf. Syst.* 106, 5 (2023), 959–975. doi:10.1587/TRANSINF.2022EDP7098

[14] Loris D'Antoni and Margus Veanes. 2014. Minimization of symbolic automata. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, Suresh Jagannathan and Peter Sewell (Eds.). ACM, 541–554. doi:10.1145/2535838.2535849

[15] James C. Davis. 2019. Rethinking Regex engines to address ReDoS. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*, Marlon Dumas, Dietmar Pfahl, Sven Apel, and Alessandra Russo (Eds.). ACM, 1256–1258. doi:10.1145/3338906.3342509

[16] James C. Davis. 2020. *On the Impact and Defeat of Regular Expression Denial of Service*. Ph. D. Dissertation. Virginia Tech, Blacksburg, VA, USA. http://hdl.handle.net/10919/98593

[17] James C. Davis, Louis G. Michael IV, Christy A. Coghlan, Francisco Servant, and Dongyoon Lee. 2019. Why aren't regular expressions a lingua franca? an empirical study on the re-use and portability of regular expressions. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*, Marlon Dumas, Dietmar Pfahl, Sven Apel, and Alessandra Russo (Eds.). ACM, 443–454. doi:10.1145/3338906.3338909

[18] Stéphane Demri and Ranko Lazić. 2009. LTL with the freeze quantifier and register automata. *ACM Transactions on Computational Logic* 10, 3 (April 2009), 1–30. doi:10.1145/1507244.1507246

[19] V8 JavaScript Engine. 2024. V8 regular expression source code. Online. Available: https://github.com/v8/v8/tree/main/src/regexp, [Accessed: 2024-11-17].

[20] Javier Esparza and Michael Blondin. 2023. *Automata Theory: An Algorithmic Approach*. MIT Press.

[21] M. Haertel et al. 2022. GNU grep. Online. Available: https://www.gnu.org/software/grep/, [Accessed: 2024-11-17].

[22] Stack Exchange. 2016. Outage Postmortem. http://stackstatus.net/post/147710624694/outage-postmortem-july-20-2016.

[23] Diego Figueira. 2012. Alternating register automata on finite words and trees. *Logical Methods in Computer Science* 8, 1 (March 2012), 22. doi:10.2168/LMCS-8(1:22)2012

[24] Dominik D. Freydenberger. 2013. Extended Regular Expressions: Succinctness and Decidability. *Theory Comput. Syst.* 53, 2 (2013), 159–193. doi:10.1007/S00224-012-9389-0

[25] Dominik D. Freydenberger and Markus L. Schmid. 2019. Deterministic regular expressions with back-references. *J. Comput. System Sci.* 105 (2019), 1–39. doi:10.1016/j.jcss.2019.04.001

[26] Bharat Garhewal, Frits Vaandrager, Falk Howar, Timo Schrijvers, Toon Lenaerts, and Rob Smits. 2020. Grey-Box Learning of Register Automata. In *Integrated Formal Methods (Lecture Notes in Computer Science)*, Brijesh Dongol and Elena Troubitsyna (Eds.). Springer International Publishing, Cham, 22–40. doi:10.1007/978-3-030-63461-2_2

[27] Google. 2022. *RE2 Issue Tracker: Feature request #101: Add backreference support.* https://github.com/google/re2/issues/101

[28] Google. 2025. *RE2*. https://github.com/google/re2

[29] Sk Adnan Hassan, Zainab Aamir, Dongyoon Lee, James C. Davis, and Francisco Servant. 2023. Improving Developers' Understanding of Regex Denial of Service Tools through Anti-Patterns and Fix Strategies. In *44th IEEE Symposium on Security and Privacy, SP 2023, San Francisco, CA, USA, May 21-25, 2023*. IEEE, 1238–1255. doi:10.1109/SP46215.2023.10179442

[30] Vojtěch Havlena, Lukáš Holík, Ondřej Lengál, Jan Vašák, and Sabína Gulčíková. 2026. Towards Efficient Matching of Regexes with Backreferences using Register Set Automata (Technical Report). *CoRR* 2205.12114 (2026). arXiv:2205.12114 https://arxiv.org/abs/2205.12114

[31] Vojtěch Havlena, Lukáš Holík, Jan Vašák Ondřej Lengál, and Sabína Gulčíková. 2026. *Artifact for the PLDI'26 paper "Towards Efficient Matching of Regexes with Backreferences using Register Set Automata"*. doi:10.5281/zenodo.19223981

[32] P. Hazel. 2024. Perl-compatible Regular Expressions. Online. Available: https://www.pcre.org/, [Accessed: 2024-11-17].

[33] Lukáš Holík, Ondřej Lengál, Olli Saarikivi, Lenka Turoňová, Margus Veanes, and Tomáš Vojnar. 2019. Succinct Determinisation of Counting Automata via Sphere Construction. In *Programming Languages and Systems (Lecture Notes in Computer Science)*, Anthony Widjaja Lin (Ed.). Springer International Publishing, Cham, 468–489. doi:10.1007/978-3-030-34175-6_24

[34] Intel. 2022. *HyperScan manual, unsupported features*. https://intel.github.io/hyperscan/dev-reference/compilation.html#unsupported-constructs
[35] Radu Iosif and Xiao Xu. 2019. Alternating Automata Modulo First Order Theories. In *Computer Aided Verification (Lecture Notes in Computer Science, Vol. 11562)*. Springer International Publishing, Cham, 43–63. doi:10.1007/978-3-030-25543-5_3
[36] Malte Isberner, Falk Howar, and Bernhard Steffen. 2014. Learning register automata: from languages to program structures. *Machine Learning* 96, 1 (July 2014), 65–98. doi:10.1007/s10994-013-5419-7
[37] Michael Kaminski and Nissim Francez. 1994. Finite-memory automata. *Theoretical Computer Science* 134, 2 (Nov. 1994), 329–363. doi:10.1016/0304-3975(94)90242-9
[38] James Kirrage, Asiri Rathnayake, and Hayo Thielecke. 2013. Static Analysis for Regular Expression Denial-of-Service Attacks. In *Network and System Security - 7th International Conference, NSS 2013, Madrid, Spain, June 3-4, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 7873)*, Javier López, Xinyi Huang, and Ravi S. Sandhu (Eds.). Springer, 135–148. doi:10.1007/978-3-642-38631-2_11
[39] Lingkun Kong, Qixuan Yu, Agnishom Chattopadhyay, Alexis Le Glaunec, Yi Huang, Konstantinos Mamouras, and Kaiyuan Yang. 2022. Software-hardware codesign for efficient in-memory regular pattern matching. In *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*, Ranjit Jhala and Isil Dillig (Eds.). ACM, 733–748. doi:10.1145/3519939.3523456
[40] Yinxi Liu, Mingxue Zhang, and Wei Meng. 2021. Revealer: Detecting and Exploiting Regular Expression Denial-of-Service Vulnerabilities. In *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*. IEEE, 1468–1484. doi:10.1109/SP40001.2021.00062
[41] F. Lundh and A. M. Kuchling. 2023. Python Standard Library: re module. Online. Available: https://docs.python.org/3/library/re.html, [Accessed: 2024-11-17].
[42] Denis Matousek, Jirí Matousek, and Jan Korenek. 2018. High-Speed Regular Expression Matching with Pipelined Memory-Based Automata. In *26th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM 2018, Boulder, CO, USA, April 29 - May 1, 2018*. IEEE Computer Society, 214. doi:10.1109/FCCM.2018.00048
[43] Robert McLaughlin, Fabio Pagani, Noah Spahn, Christopher Kruegel, and Giovanni Vigna. 2022. Regulator: Dynamic Analysis to Detect ReDoS. In *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*, Kevin R. B. Butler and Kurt Thomas (Eds.). USENIX Association, 4219–4235. https://www.usenix.org/conference/usenixsecurity22/presentation/mclaughlin
[44] Microsoft. 2024. .NET 8.0.110 regular expressions in System.Text.RegularExpressions. Online. Available: https://learn.microsoft.com/en-us/dotnet/api/system.text.regularexpressions?view=net-8.0, [Accessed: 2024-11-17].
[45] Marvin L. Minsky. 1961. Recursive Unsolvability of Post's Problem of "Tag" and other Topics in Theory of Turing Machines. *Annals of Mathematics* 74, 3 (1961), 437–455. doi:10.2307/1970290
[46] Andrzej S. Murawski, Steven J. Ramsay, and Nikos Tzevelekos. 2018. Polynomial-Time Equivalence Testing for Deterministic Fresh-Register Automata. (2018), 14 pages. doi:10.4230/LIPICS.MFCS.2018.72
[47] Kedar S. Namjoshi and Girija J. Narlikar. 2010. Robust and Fast Pattern Matching for Intrusion Detection. In *INFOCOM'10*. IEEE, 740–748. doi:10.1109/INFCOM.2010.5462149
[48] Frank Neven, Thomas Schwentick, and Victor Vianu. 2004. Finite state machines for strings over infinite alphabets. *ACM Trans. Comput. Log.* 5, 3 (2004), 403–435. doi:10.1145/1013560.1013562
[49] Taisei Nogami and Tachio Terauchi. 2025. Efficient Matching of Some Fundamental Regular Expressions with Backreferences. In *50th International Symposium on Mathematical Foundations of Computer Science, MFCS 2025, August 25-29, 2025, Warsaw, Poland (LIPIcs, Vol. 345)*, Pawel Gawrychowski, Filip Mazowiecki, and Michal Skrzypczak (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 81:1–81:19. doi:10.4230/LIPICS.MFCS.2025.81
[50] OpenJDK. 2024. JDK 17 API Documentation: Regular Expressions. Online. Available: https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/regex/package-summary.html, [Accessed: 2024-11-17].
[51] Francesco Parolini and Antoine Miné. 2022. Sound Static Analysis of Regular Expressions for Vulnerabilities to Denial of Service Attacks. In *Theoretical Aspects of Software Engineering - 16th International Symposium, TASE 2022, Cluj-Napoca, Romania, July 8-10, 2022, Proceedings (Lecture Notes in Computer Science, Vol. 13299)*, Yamine Aït Ameur and Florin Craciun (Eds.). Springer, 73–91. doi:10.1007/978-3-031-10363-6_6
[52] Michael O. Rabin and Dana S. Scott. 1959. Finite Automata and Their Decision Problems. *IBM J. Res. Dev.* 3, 2 (1959), 114–125. doi:10.1147/rd.32.0114
[53] Regex101.com. 2025. https://regex101.com. [Online; accessed 25-March-2025].
[54] M. Roesch et al. 2022. *Snort: A Network Intrusion Detection and Prevention System,*. Cisco. http://www.snort.org
[55] Hiroshi Sakamoto and Daisuke Ikeda. 2000. Intractability of decision problems for finite-memory automata. *Theor. Comput. Sci.* 231, 2 (2000), 297–308. doi:10.1016/S0304-3975(99)00105-X

[56] Markus L Schmid. 2016. Characterising REGEX Languages by Regular Languages Equipped with Factor-Referencing. 249 (Aug. 2016), 1–17. doi:10.1016/j.ic.2016.02.003

[57] Sylvain Schmitz. 2016. Complexity Hierarchies Beyond Elementary. *ACM Transactions on Computation Theory* 8, 1 (Feb. 2016), 1–36. doi:10.1145/2858784

[58] Sylvain Schmitz. 2017. *Algorithmic Complexity of Well-Quasi-Orders.* Habilitation à diriger des recherches. École normale supérieure Paris-Saclay. https://tel.archives-ouvertes.fr/tel-01663266

[59] Sylvain Schmitz and Philippe Schnoebelen. 2012. Algorithmic Aspects of WQO Theory. (Aug. 2012). https://cel.archives-ouvertes.fr/cel-00727025

[60] Sylvain Schmitz and Philippe Schnoebelen. 2013. The Power of Well-Structured Systems. In *CONCUR 2013, Buenos Aires, Argentina, August 27-30, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 8052)*, Pedro R. D'Argenio and Hernán C. Melgratti (Eds.). Springer, 5–24. doi:10.1007/978-3-642-40184-8_2

[61] Luc Segoufin. 2006. Automata and Logics for Words and Trees over an Infinite Alphabet. In *Computer Science Logic*. Vol. 4207. Springer Berlin Heidelberg, Berlin, Heidelberg, 41–57. doi:10.1007/11874683_3

[62] Yuju Shen, Yanyan Jiang, Chang Xu, Ping Yu, Xiaoxing Ma, and Jian Lu. 2018. ReScue: crafting regular expression DoS attacks. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, Marianne Huchard, Christian Kästner, and Gordon Fraser (Eds.). ACM, 225–235. doi:10.1145/3238147.3238159

[63] Weihao Su, Hong Huang, Rongchen Li, Haiming Chen, and Tingjian Ge. 2024. Towards an Effective Method of ReDoS Detection for Non-backtracking Engines. In *33rd USENIX Security Symposium, USENIX Security 2024, Philadelphia, PA, USA, August 14-16, 2024*, Davide Balzarotti and Wenyuan Xu (Eds.). USENIX Association. https://www.usenix.org/conference/usenixsecurity24/presentation/su-weihao

[64] Nvidia Mellanox team. 2021. Personal communication.

[65] Tachio Terauchi. 2025. On DoS Vulnerability of Regular Expressions, with and Without Backreferences. In *38th IEEE Computer Security Foundations Symposium, CSF 2025, Santa Cruz, CA, USA, June 16-20, 2025*. IEEE, 190–204. doi:10.1109/CSF64896.2025.00011

[66] Ken Thompson. 1968. Programming Techniques: Regular expression search algorithm. *Commun. ACM* 11, 6 (1968), 419–422. doi:10.1145/363347.363387

[67] Milo Tova, Suciu Dan, and Vianu Victor. 2000. Typechecking for XML transformers. In *POD'00*. ACM, Dallas, USA, 11–22. doi:10.1145/335168.335171

[68] Lenka Turoňová, Lukáš Holík, Ivan Homoliak, Ondřej Lengál, Margus Veanes, and Tomáš Vojnar. 2022. Counting in Regexes Considered Harmful: Exposing ReDoS Vulnerability of Nonbacktracking Matchers. In *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*, Kevin R. B. Butler and Kurt Thomas (Eds.). USENIX Association, 4165–4182. https://www.usenix.org/conference/usenixsecurity22/presentation/turonova

[69] Lenka Turoňová, Lukáš Holík, Ondřej Lengál, Olli Saarikivi, Margus Veanes, and Tomáš Vojnar. 2020. Regex matching with counting-set automata. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (Nov. 2020), 218:1–218:30. doi:10.1145/3428286

[70] Nikos Tzevelekos and Radu Grigore. 2013. History-Register Automata. In *Foundations of Software Science and Computation Structures*. Springer Berlin Heidelberg, Berlin, Heidelberg, 17–33.

[71] Ian Erik Varatalu, Margus Veanes, and Juhan P. Ernits. 2025. RE#: High Performance Derivative-Based Regex Matching with Intersection, Complement, and Restricted Lookarounds. *Proc. ACM Program. Lang.* 9, POPL (2025), 1–32. doi:10.1145/3704837

[72] Jan Vašák. 2026. `rsamatch`. https://github.com/VeriFIT/RegisterSetAutomata

[73] Xiang Wang, Yang Hong, Harry Chang, KyoungSoo Park, Geoff Langdale, Jiayu Hu, and Heqing Zhu. 2019. Hyperscan: A Fast Multi-pattern Regex Matcher for Modern CPUs. In *NSDI'19*, Jay R. Lorch and Minlan Yu (Eds.). USENIX Association, 631–648. https://www.usenix.org/conference/nsdi19/presentation/wang-xiang

[74] Xinyi Wang, Cen Zhang, Yeting Li, Zhiwu Xu, Shuailin Huang, Yi Liu, Yican Yao, Yang Xiao, Yanyan Zou, Yang Liu, and Wei Huo. 2023. Effective ReDoS Detection by Principled Vulnerability Modeling and Exploit Generation. In *2023 IEEE Symposium on Security and Privacy (SP)*. 2427–2443. doi:10.1109/SP46215.2023.10179328

[75] Nicolaas Weideman, Brink van der Merwe, Martin Berglund, and Bruce W. Watson. 2016. Analyzing Matching Time Behavior of Backtracking Regular Expression Matchers by Using Ambiguity of NFA. In *Implementation and Application of Automata - 21st International Conference, CIAA 2016, Seoul, South Korea, July 19-22, 2016, Proceedings (Lecture Notes in Computer Science, Vol. 9705)*, Yo-Sub Han and Kai Salomaa (Eds.). Springer, 322–334. doi:10.1007/978-3-319-40946-7_27

[76] Valentin Wüstholz, Oswaldo Olivo, Marijn J. H. Heule, and Isil Dillig. 2017. Static Detection of DoS Vulnerabilities in Programs that Use Regular Expressions. In *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 10206)*, Axel Legay and Tiziana Margaria (Eds.). 3–20. doi:10.1007/978-3-662-54580-5_1

Received 2025-11-14; accepted 2026-04-03