REDoS Detection in "Domino" Regular Expressions by Ambiguity Analysis

Antonina Nepeivoda
Aylamazyan Program Systems Institute of RAS
Pereslavl–Zalesskiy
ORCID 0000-0003-3949-2164

Yulia Belikova
Bauman Moscow State Technical University
Moscow
yulyawh@gmail.com

Kirill Shevchenko
Bauman Moscow State Technical University
Moscow
k.sh3vch3nko@yandex.ru

Mikhail Teriukha
Bauman Moscow State Technical University
Moscow
misha37a999@yandex.ru

Danila Knyazihin
Bauman Moscow State Technical University
Moscow
dak151449@gmail.com

Aleksandr Delman
Bauman Moscow State Technical University
Moscow
adelman2112@gmail.com

Anna Terentyeva
Bauman Moscow State Technical University
Moscow
mathhyyn@gmail.com

***Abstract*—The Regular Expression Denial of Service (REDoS) problem refers to a time explosion caused by the high computational complexity of matching a string against a regex pattern. This issue is prevalent in popular regex engines, such as PYTHON, JAVASCRIPT, and C++. In this paper, we examine several existing open-source tools for detecting REDoS and identify a class of regexes that can create REDoS situations in popular regex engines but are not detected by these tools.**

To address this gap, we propose a new approach based on ambiguity analysis, which combines a strong star-normal form test with an analysis of the transformation monoids of Glushkov automata orbits. Our experiments demonstrate that our implementation outperforms the existing tools on regexes with polynomial matching complexity and complex subexpression overlap structures.

***Index Terms*—regular expressions, ambiguity, REDoS, Glushkov automaton, transformation monoid, strong star-normal form**

I. INTRODUCTION

Popular regular expression (regex) engines typically use non-deterministic finite automata (NFA) as their internal representation for regexes. This choice is motivated by the flexibility of the NFA concept, which can be extended to support a wider range of regex operations with little effort. For instance, back-references and lookaheads can be easily added to the NFA model. Although, in theory, every string can be matched against a regex in linear time using deterministic finite automata (DFA) conversion, popular regex engines may admit exponential matching time due to a phenomenon called "catastrophic backtracking".

This phenomenon occurs only for a specific class of regular expressions. For example, consider the regex $(\mathtt{a} \mid \mathtt{b})^*\mathtt{a}$, which is non-deterministic due to the unavoidable non-determinism in the transition to the last occurrence of the letter a. However, every string has a unique parsing tree with respect to this regex. In contrast, the regex $(\mathtt{a}^*\mathtt{b}^*)^*$ has an infinite number of accepting parsing trees for any given string, as inner Kleene stars can degenerate to the empty word, causing a combinatorial explosion of parse paths. Intuitively, the latter regex can be considered "bad", while the former is considered "good".

Matching against "bad" regexes can yield a situation called a Regular Expression Denial of Service (REDoS), when the matching time grows super-linearly and can cause performance issues in, for instance, a web service that uses such a regex to parse user input. To avoid these situations, it is essential to detect unsafe regexes and replace them with safe equivalents.

The number of research papers mentioning the REDoS problem has increased rapidly in the last decade [1]–[7]. Several tools have been developed to detect REDoS, using both static analysis and random search. Some of these tools aim to detect the entire class of extended regexes, while others focus on academic ones. However, for a class of simple regexes, which are not safe in theory, the tools considered either take too long time to process, or give an incorrect answer, falsely witnessing their safety. These regexes usually have overlapping, but not completely coinciding, structure of the expressions under the Kleene stars (being a simple analogue of

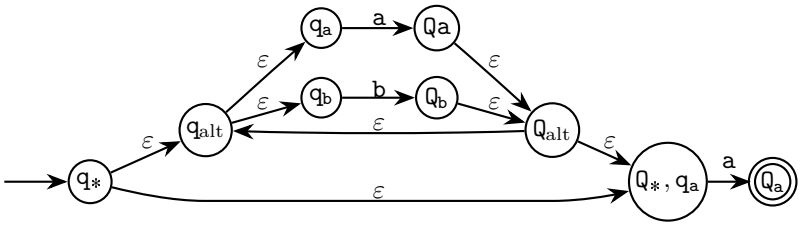


Fig. 1: Thompson automaton for $(\mathtt{a} \mid \mathtt{b})^*\mathtt{a}$

dominoes in the Post Correspondence Problem). An example of such a regex is $(\mathtt{baa} \mid \mathtt{ab})^*(\mathtt{b} \mid \varepsilon)(\mathtt{a}(\mathtt{ba} \mid \mathtt{a})\mathtt{ba}^*\mathtt{b})^*(\mathtt{aab})^*$: the ambiguity occurs both in prefixes $(\mathtt{baa})^n$ and $(\mathtt{ab})^n$, which can be constructed in several ways from primitive "dominoes".

Thus, the two natural research questions arise:

- do the "domino" regexes really contain REDoS situations w.r.t. the modern regex engines?
- if the answer is yes, what methods can deal with such regexes in order to analyse them without blow-up of the analysis time because of the overlaps?

The main contributions of the paper are:

- a method for REDoS situations detection, utilizing properties of non-deterministic finite automata and their transition monoids. This approach is novel, since previous static-analysis-based methods use NFA intersection. For "domino" regexes our method is shown to perform better than the open-source analogues REGEX STATIC ANALYZER [3], RESCUE [5], and REVEALER [2].
- experimental testing of the relevance of the NFA model used and the vulnerabilities found, by investigating real regex engines behaviour on the attack strings.

The method is implemented only for the academic regexes for now. Surprisingly, for this case, the tested open-source tools perform significantly worse on domino tests, especially for polynomial REDoS situations.

The paper is organized as follows. Section II contains preliminaries on finite automata, and theoretical concepts that are used further. The proposed REDoS detection method is given in Section III, preceded by lemmas used for its optimisation. Section IV discusses relevance of the chosen model with respect to the real regex matching engines, and provides a result of comparative testing of our method and three other open-source REDoS detection tools. We discuss the results of the experiments and the related works in more detail in Section V. Section VI concludes the paper.

II. PRELIMINARIES

We denote automata with calligraphic $\mathscr{A}$; states are denoted with the letters q and Q, or with the set of these letters (if an automaton is a result of a closure operation). The empty word is denoted by ε; concrete elements from the input alphabet are denoted with $\mathtt{a}$, $\mathtt{b}$, $\mathtt{c}$, ..., and letter parameters are denoted with γ; ω and η denote word parameters. We use only the basic academic regular expression constructing operations: concatenation (which is omitted in notation), alternation (denoted with $|$), and Kleene star (denoted with *). If r is a regex, $\mathscr{L}(r)$ denotes its language.

Let us recall basic definitions and describe the finite automata models used in this paper.

A. Finite Automata

Definition II.1. *A non-deterministic finite automaton (NFA) is a tuple $\langle S, \Sigma, q_0, F, \delta\rangle$, where:*

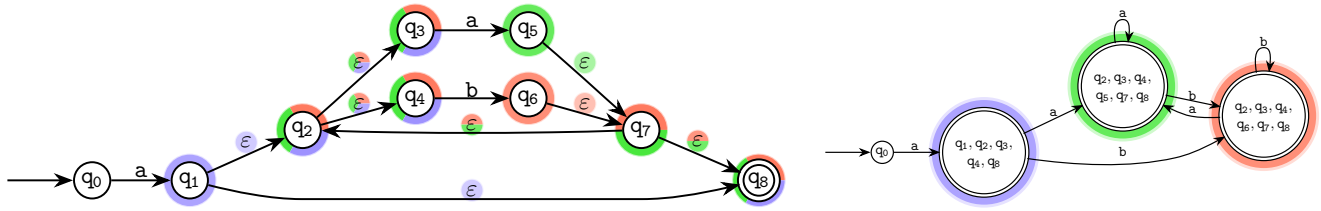
- *S is a state set;*
- *Σ is a terminal alphabet;*
- *δ is a set of transitions of the form $\langle q_i, (\gamma_i|\varepsilon), M_i\rangle$, where $q_i \in S$, $\gamma_i \in \Sigma$, $M_i \in 2^S$;*
- *$q_0 \in S$ is the initial state;*
- *$F \subseteq S$ is a set of final states.*

Every transition in an NFA maps a pair $\langle q_i, (\gamma_i|\varepsilon)\rangle$ into a set of states, contrary to transitions in a deterministic finite automaton (DFA), which map every pair $\langle q_i, \gamma_i\rangle$ (where γ_i is essentially not equal to ε) to a single state. Thus, if a word is parsed by a DFA, the parse trace is always unique (i.e., DFAs are unambiguous); in an NFA, there can be a set of parse traces for a single word. This set can even be infinite in case of NFA with ε-transitions. The notation $q_i \xrightarrow{\gamma} \ldots$ is overloaded to denote either NFA transition $\langle q_i, \gamma, M_i\rangle$ (written as $q_i \xrightarrow{\gamma} M_i$) or a transition to a single state belonging to M_i (written as $q_i \xrightarrow{\gamma} q_j$). Existence of a path from q_i to q_j marked by $\omega \in \Sigma^*$ is denoted by $q_i \xrightarrow{\omega}\!\!\!\!\rightarrow q_j$.

An NFA can be transformed into an equivalent DFA using a textbook subset-constructing algorithm `Determinize`, which generates states of the DFA corresponding to the sets of the states of the initial NFA resulted in the transitions along the same input symbols.

The NFA models used in regex engines are primarily based on the classical Thompson construction, which provides an algorithm for transforming a regex into an NFA that recognizes the same language. While the implementation details of the transformation may vary, the experiments presented in Section IV provide evidence that the Thompson model remains relevant for identifying inefficient regexes with respect to NFA-based parsing engines.

In the following descriptions, we only give details of the constructed NFAs in terms of their states and transitions, without mentioning the alphabet construction.



(a) $\texttt{Thompson}(\texttt{a}(\texttt{a} \mid \texttt{b})^*)$ with colored ε-closures (b) automaton after merging ε-closures

Fig. 2: Constructing $\texttt{Glushkov}(r)$ based on $\texttt{Thompson}(r)$

Definition II.2. *Thompson NFA (denoted with* $\texttt{Thompson}(r)$*) is constructed from a regex* r *as follows. At any construction step except processing concatenations, the new initial state* q_r *and the new final state* Q_r *are introduced, and the transition set is updated depending on the regex operation.*

- *Every single letter* γ *generates a primitive automaton with the only transition* $q_\gamma \xrightarrow{\gamma} \{Q_\gamma\}$.
- *If* $\mathscr{A}_1 = \texttt{Thompson}(r_1)$, $\mathscr{A}_2 = \texttt{Thompson}(r_2)$, *and* q_i *and* Q_i *are their initial and final states, respectively, then* $\texttt{Thompson}(r_1 \mid r_2)$ *is constructed by merging the* $\mathscr{A}_1$ *and* $\mathscr{A}_2$ *states sets and transitions sets, and introducing the transitions* $q_{\text{alt}} \xrightarrow{\varepsilon} \{q_1, q_2\}$; $Q_1 \xrightarrow{\varepsilon} \{Q_{\text{alt}}\}$; $Q_2 \xrightarrow{\varepsilon} \{Q_{\text{alt}}\}$.
- $\texttt{Thompson}(r_1\, r_2)$ *is again constructed by merging* $\texttt{Thompson}(r_i)$ *states and transitions sets, and making* q_1 *the initial state,* Q_2 *the final state, with the additional transition* $Q_1 \xrightarrow{\varepsilon} \{q_2\}$.
- $\texttt{Thompson}(r_1^*)$ *is constructed introducing transitions* $q_* \xrightarrow{\varepsilon} \{q_1, Q_*\}$, $Q_1 \xrightarrow{\varepsilon} \{q_1, Q_*\}$.

The Thompson construction algorithm ensures that any NFA produced by the algorithm has a unique final state, and that each state has at most two outgoing and two incoming transition arcs. The uniqueness of the final state implies that the reverse NFA for $\texttt{Thompson}(r)$ is exactly $\texttt{Thompson}(r^R)$, where r^R is the reverse of the regex r. Additionally, all subregex automata can be treated as isolated directed acyclic graphs, which makes the construction easily extensible and decomposable. An example of a Thompson automaton for a regex is shown in Figure 1. The states labels follow the corresponding regex operations given in Definition II.2.

One drawback of the Thompson construction is that it introduces non-deterministic transitions corresponding to alternating operations (i.e., alternatives or Kleene stars), even in the cases when the regex itself imposes no non-determinism (e.g. for the regex $\texttt{a}(\texttt{a} \mid \texttt{b})^*$, which is a reverse of the regex shown in Fig. 1). To avoid the redundant non-determinism, the regex engine RE2 [8] processes such strongly deterministic regexes (also known as 1-unambiguous regexes [9]) constructing another NFA based on the regex structure, but without ε-transitions. This automaton is known as the Glushkov automaton since 1960s, and in the last two decades it attracted considerable interest, shown to be efficient and extensible to construct deterministic parsing engines for a larger class of regexes (such as memory finite automata for the regexes with back-references [10]).

The classical `Glushkov` construction is based on so-called follow-relation on linearised regexes. By construction, every state in the Glushkov automaton except the initial state corresponds to an occurrence of some $\gamma \in \Sigma$ in the input regex r; conversely, any letter occurrence in the regex r corresponds to exactly one state in $\texttt{Glushkov}(r)$, whose incoming arcs are all marked with γ. Now we can reformulate this property in the terms of Thompson and Glushkov automata.

Proposition II.1. *There is a bijection from state set in* $\texttt{Glushkov}(r)$ *minus the initial state to state set* Q_γ *in* $\texttt{Thompson}(r)$ *(where* Q_γ *are final states of the primitive automata reading* γ*).*

In the paper [11], it was shown that $\texttt{Glushkov}(r)$ can be also obtained from $\texttt{Thompson}(r)$ merging its ε-closures.

Definition II.3. *Given an NFA* $\mathscr{A}$ *and its state* q*,* ε*-closure of* q *is the maximal set of states reachable from* q *following only* ε*-transitions.*

Closure-merging[1] *ε-free automaton (denoted with* $\texttt{RemEps}(\mathscr{A})$*) is constructed from* $\mathscr{A}$ *as follows:*

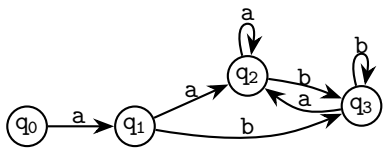
- *its states are* ε*-closures of the states of* $\mathscr{A}$*;*
- *if state* q_1 *belongs to closure* C_i*, state* q_2 *belongs to* C_j*, and there is a transition* $q_1 \xrightarrow{\gamma} \{..., q_2, ...\}$ $(\gamma \neq \varepsilon)$ *in* $\mathscr{A}$*, then there is a transition* $C_i \xrightarrow{\gamma} \{..., C_j, ...\}$ *in* $\texttt{RemEps}(\mathscr{A})$.

An example of closure-merging operation is given in Fig. 2.

B. Transformation Monoid of NFA

Let us consider an automaton with no useless states and ε-transitions. Its transitions over the alphabet Σ and the states set 2^Q form the function $F : \Sigma \times S \to 2^S$ taking a pair $\langle \gamma,\ q_i \rangle$. This function, when curried and specialized in the first argument, becomes $F_\gamma : S \to 2^S$ (where $\gamma \in \Sigma$). We can form a monoid over the set of such partially specialized functions (transformations) if we continue them on strings as follows: $F_{\omega_1} \circ F_{\omega_2} = F_{\omega_2 \omega_1}$. Then associativity is provided "for free", given associativity of string concatenation, and ε becomes the monoid unit, because $F_\omega \circ F_\varepsilon = F_{\varepsilon\omega} = F_\omega = F_{\omega\varepsilon} = F_\varepsilon \circ F_\omega$

[1] This ε-removal construction differs from the standard textbook ε-removal algorithm, since it changes states, and not only transitions. This strategy allows the algorithm to succeed in conversion from `Thompson` to `Glushkov`.



(a) Labelled transition system of $\mathscr{A}$

	q_0	q_1	q_2	q_3
a	$\{q_1\}$	$\{q_2\}$	$\{q_2\}$	$\{q_2\}$
b	$\{\}$	$\{q_3\}$	$\{q_3\}$	$\{q_3\}$
aa	$\{q_2\}$	$\{q_2\}$	$\{q_2\}$	$\{q_2\}$
ab	$\{q_3\}$	$\{q_3\}$	$\{q_3\}$	$\{q_3\}$
ba	$\{\}$	$\{q_2\}$	$\{q_2\}$	$\{q_2\}$

(b) Equivalence classes

$\mathtt{bb} \mapsto \mathtt{b}$ $\quad \mathtt{aaa} \mapsto \mathtt{aa}$
$\mathtt{aab} \mapsto \mathtt{ab}$ $\quad \mathtt{aba} \mapsto \mathtt{aa}$
$\mathtt{baa} \mapsto \mathtt{ba}$ $\quad \mathtt{bab} \mapsto \mathtt{bb}$

(c) Rewriting rules

Fig. 3: Transformation monoid of $\mathscr{A} = \mathtt{Glushkov(a(a \mid b)^*)}$

holds. The state transformations are denoted by the corresponding strings ω.

The formal definition is as follows [12].

Definition II.4. *Given an ε-free automaton $\mathscr{A}$ over the alphabet Σ^*, its transformation monoid $\mathscr{M} = \mathtt{TransMonoid}(\mathscr{A})$ is the monoid of transformations imposed by elements of Σ^* on the states of $\mathscr{A}$.*

The monoid construction does not depend on the choice of the final or initial states of $\mathscr{A}$ (except the condition that all the states are useful, i.e. reachable and producing), thus, instead of classical NFAs, the monoid is based on a labelled transition system. Since the set of functions $S \to 2^S$ is finite, the transformation monoid of an NFA always contains a finite number of equivalence classes. The pair $\langle M,\ R\rangle$, where M is a finite set of lexicographically minimal elements of the equivalence classes and R is a set of simplification rules is considered a standard representation of the transformation monoid. Such a representation for $\mathtt{TransMonoid(Glushkov(a(a \mid b)^*))}$ is given in Fig. 3. The monoid representation uncovers some useful NFA properties. For example, we can immediately conclude that the words aa and ab are synchronizing, since for all q_i, $q_i \xrightarrow{\mathtt{aa}\!\!\rightarrow} q_2$, $q_i \xrightarrow{\mathtt{ab}\!\!\rightarrow} q_3$, and no other transition is possible.

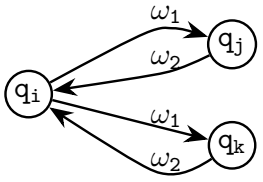
C. Ambiguity of NFAs and REDoS

Intuitively, the worst-case scenario for backtracking-based matching of a string against a regex r occurs when the matched string has a prefix η_1 with a large set of parse paths, and a suffix η_2 s.t. $\eta_1\eta_2 \notin \mathscr{L}(r)$. In this case, in order to determine that $\eta_1\eta_2$ is not recognizable by r, a regex engine must backtrack through all the parse variants of η_1. Obviously, we can choose such a suffix η_3 that $\eta_1\eta_3 \in \mathscr{L}(r)$, and $\eta_1\eta_3$ will still have a large number of parse trees (although the regex engine will report a success after finding a first one). Therefore, worst-case matching time depends on the upper bound on the parse paths in a regex.

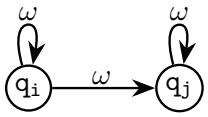
In the domain of finite automata, the following definition is used [13], [14].

Definition II.5. *A degree of ambiguity of an NFA $\mathscr{A}$ is a worst-case bound on the number of paths recognizing an input string (in a length of the string).*

The ambiguity of NFAs is known to be either a constant, an exponential, or a polynomial [13]. If the ambiguity degree of $\mathscr{A}$ is non-constant, it is said $\mathscr{A}$ has an infinite degree of ambiguity (IDA). A standard acronym for exponential ambiguity degree is EDA.



(a) Exponential ambiguity
($\mathtt{q_j}$ or $\mathtt{q_k}$ may coincide with $\mathtt{q_i}$)



(b) Polynomial ambiguity

Fig. 4: Ambiguity situations in NFA

A minimal EDA-generating regex example is $(\mathtt{a} \mid \mathtt{a})^*$. A minimal example of a regex producing IDA but not EDA automaton is $\mathtt{a^*a^*}$. For regexes s.t. $(\mathtt{a^*b^*})^*$, $\mathtt{Glushkov}(r)$ is unambiguous, despite $\mathtt{Thompson}(r)$ is EDA. We can notice that in $\mathtt{Thompson}((\mathtt{a^*b^*})^*)$, a special situation occurs: there is a loop inside an ε-closure of a state (i.e., there is at least one Kleene star in a regex iterating over an expression r_E s.t. $\varepsilon \in \mathscr{L}(r_E)$). Further we show that such a case is the only possible situation when $\mathtt{Thompson}(r)$ and $\mathtt{Glushkov}(r)$ have distinct ambiguity degrees.

The following criterion estimates the degree of ambiguity in any NFA.

Theorem II.2.
- *$\mathscr{A}$ satisfies IDA condition $\Leftrightarrow$ there exist states q_i, q_j in $\mathscr{A}$, and a word ω s.t. $\mathscr{A}$ contains paths from q_i and q_j to themselves, and a path from q_i to q_j all accepting the word ω.*
- *$\mathscr{A}$ satisfies EDA condition $\Leftrightarrow$ there exists a state q_i in $\mathscr{A}$ and a word ω s.t. $\mathscr{A}$ contains two distinct loops from q_i to itself both accepting the word ω.*

We can also say than if EDA occurs in an NFA, then $\exists q_i, q_j, q_k, \omega_1, \omega_2 (q_i \xrightarrow{\omega_1\!\!\rightarrow} q_j \ \&\ q_i \xrightarrow{\omega_1\!\!\rightarrow} q_k \ \&\ q_j \xrightarrow{\omega_2\!\!\rightarrow} q_i \ \&\ q_k \xrightarrow{\omega_2\!\!\rightarrow} q_i)$ (see Fig. 4).

After the work [9], we use the term "orbit of state q" for the maximal strongly connected component containing q. We assume that orbits are non-trivial, i.e. contain at least one transition. If a state q of $\mathscr{A}$ satisfies EDA criterion for some ω, then all states belonging to its orbit also satisfy EDA. Thus, to check the EDA condition, it is sufficient to check if any state of some strongly connected component of an NFA satisfies

EDA; for the IDA condition, it is sufficient to check if there are two strongly connected components satisfying it.

An approach to the IDA and EDA detection used in the REDoS analysers [3], [4] tests the above criterion constructing single or double intersections of $\mathscr{A}$ with itself. Although the intersection construction can be done in polynomial time on an NFA size, it may lead to large NFAs if there are many crossing components (i.e., matching the same string sets) in the initial NFA.

The IDA criterion can be also reformulated in the terms of transformation monoids.

Proposition II.3. *An ε-free $\mathscr{A}$ satisfies IDA $\Leftrightarrow$ its transformation monoid contains an equivalence class ω s.t. for some states q_i, q_j, $q_i \in F_\omega(q_i)$, $q_j \in F_\omega(q_j)$, and $q_j \in F_\omega(q_i)$.*

Using this criterion for an initial NFA "as is" is highly impractical: if the NFA contains non-crossing components, the transformation monoid becomes exponentially huge. However, with some refinements, we observed that the monoid criterion can be applied (and even be fast) in the cases when the intersection criterion is slow. Moreover, Proposition II.3 provides explicit construction of a string with the ambiguity, allowing the analysing algorithm to reconstruct the REDoS situation easily. First, take any NFA path from the initial state of $\mathscr{A}$ to q_i, recognizing some prefix η_1. Then pump ω to construct an infix with superlinear number of parse trees, and then take some string η_2 s.t. any path from q_j recognizing η_2 does not end in a final state of $\mathscr{A}$. The string $\eta_1\omega^n\eta_2$ will force an NFA parsing device to do superlinear backtracking.

If the monoid criterion is applied to the orbit automaton of state q, the REDoS pump can be constructed as well. Just choose some η_1, η_2 s.t. $q_0 \xrightarrow{\eta_1} q$, $\forall q_F \in F(\neg\, q \xrightarrow{\eta_2} q_F)$.

III. Our Approach

As a starting point, we prefer to use the Thompson automaton as a preliminary NFA model for a regex since regex matching engines rely on it in their internal algorithms, and experiments in Section IV demonstrate that the Thompson construction is suitable for analysing real REDoS. However, in order to apply the monoid criterion, we must first eliminate ε-transitions in the regex and ensure that the removal of ε-transitions does not affect the degree of ambiguity.

Let us say that regex r is in a (strong) star-normal form (SSNF) if it does not contain a subexpression $(r_0)^*$ s.t. $\varepsilon \in \mathscr{L}(r_0)$ [15]. The following proposition gives an equivalent criterion.

Proposition III.1. *r is in SSNF $\Leftrightarrow$ no ε-closure of* $\texttt{Thompson}(r)$ *contains a loop.*

Proof. $\Leftarrow$: Let r contain a subexpression $(r_E)^*$, where $\varepsilon \in \mathscr{L}(r_E)$, and let q_E and Q_E be the initial and final states of $\texttt{Thompson}(r_E)$. Since there is a path in $\texttt{Thompson}(r)$ from q_E to Q_E recognizing ε, ε-closure of q_E contains a loop.

$\Rightarrow$: Any loop must contain a backward arc, and in any Thompson automaton, the only backward arcs are transitions from Q_E to q_E, where r_E is a subexpression under a Kleene star. If the loop follows ε-transitions, it also contains a path $q_E \xrightarrow{\varepsilon} Q_E$, so $\varepsilon \in \mathscr{L}(r_E)$, and $(r_E)^*$ breaks the SSNF condition. □

In the following proposition, `Ambiguity` is valued either EDA, IDA (not EDA), or safe.

Proposition III.2. *If r is in the SSNF, then* $\texttt{Ambiguity}(\texttt{Thompson}(r)) = \texttt{Ambiguity}(\texttt{Glushkov}(r))$.

Proof. Any strongly connected component in $\texttt{Thompson}(r)$, as well as in $\texttt{Glushkov}(r)$, corresponds to a subexpression r' under a Kleene star in r (by construction). If an IDA occurs in this subexpression in `Thompson`, say, for a state q and word $\omega \neq \varepsilon$, then there exist two distinct states Q_γ, Q'_γ, which are final states of primitive automata for letter $\gamma \in \Sigma$, and there are paths $q \xrightarrow{\omega_1} Q_\gamma$, $q \xrightarrow{\omega_1} Q'_\gamma$, $Q_\gamma \xrightarrow{\omega_2} q$, $Q'_\gamma \xrightarrow{\omega_2} q$ s.t. $\omega = \omega_1\omega_2$, $\omega_1 \neq \varepsilon$. Thus, γ occurs twice in r', hence these occurrences correspond to distinct states of the Glushkov automaton. Therefore, the paths $\texttt{EpsClosure}(q) \xrightarrow{\omega_1} \texttt{EpsClosure}(Q_\gamma)$ and $\texttt{EpsClosure}(q) \xrightarrow{\omega_1} \texttt{EpsClosure}(Q'_\gamma)$ are distinct, and there is the EDA situation in $\texttt{EpsClosure}(q)$.

If $\texttt{Thompson}(r)$ contains an IDA which is not an EDA, then the IDA-producing states belong to distinct Kleene star subexpressions. Moreover, since $\omega \neq \varepsilon$, in the paths producing the IDA situation, there are at least two distinct states Q_γ, Q'_γ, which are final states of primitive automata for letter $\gamma \in \Sigma$ and have distinct orbits. Thus, their ε-closures remain to be distinct. □

Thus, it is sufficient to test r for the strong star-normal form property and then, if necessary, continue the ambiguity analysis operating with the Glushkov automaton, having significantly less states. If there are loops in ε-closures, the further analysis is not needed: these loops already produce EDA situations.

Given a state q in $\mathscr{A}$ and its orbit M, an orbit automaton of q is automaton M_q including all states and transitions from M, having q as is the initial state, and whose final states are either final states of $\mathscr{A}$ or states with outgoing transitions outside the orbit M in $\mathscr{A}$.

If we choose one state q_i from each strongly connected component C_i of $\mathscr{A}$, then testing an IDA criterion for $\texttt{TransMonoid}(M_{q_i})$ is enough to reveal all EDA situations. However, in the case of a polynomial IDA, we must test pairs of the strongly connected components (together with the transitions from one component to another), and building a monoid for any such pair-generated NFA is too time-consuming. Thus, we use the following simple necessary condition for the polynomial IDA.

Proposition III.3. *Let C_1, C_2 be distinct strongly connected components of $\mathscr{A}$. If $\mathscr{A}$ contains a polynomial IDA within the components, then there exist two states, $q_1 \in C_1$, $q_2 \in C_2$, s.t.* $\texttt{Determinize}(\mathscr{A})$ *contains a subset state including both q_1 and q_2. Moreover, such a subset state occurs also in* $\texttt{Determinize}(\texttt{Reverse}(\mathscr{A}))$.

Although the determinization algorithm is exponentially hard in the worst case, it is known to be fast in most practical cases [16]. Thus, the subset test accelerates candidates search for the polynomial IDA. However, it is not sufficient, which can be shown by analysing regex $(\mathtt{a} \mid \mathtt{b})^*(\mathtt{b} \mid \mathtt{c})(\mathtt{a} \mid \mathtt{c})^*$ whose Thompson automaton contains no IDA.

A. Algorithm

if $\neg\,\mathtt{SSNF}(r)$ **then**
  return EDA
  $\mathscr{A} \leftarrow \mathtt{Glushkov}(r)$
end if
$\mathcal{C} \leftarrow \mathtt{SCC}(\mathscr{A})$
for $c \in \mathcal{C}$ **do**
  $q_0 \leftarrow c[1]$
  if $\mathtt{Ambiguity}(\mathtt{TransMonoid}(M_{q_0}))$ **then**
    return EDA
  end if
end for
for $c_1, c_2 \in \mathcal{C}$ **do**
  if $c_1 \longrightarrow\!\!\!\!\!\rightarrow c_2$ **then**
    $q_1 \leftarrow c_1[1]$
    $q_2 \leftarrow c_2[1]$
    if $\mathtt{SubsetPairs}(\mathtt{Determinize}(\mathscr{A}_{q_1+q_2})) \cap \mathtt{SubsetPairs}(\mathtt{Determinize}(\mathtt{Reverse}(\mathscr{A}_{q_1+q_2}))) \neq \varnothing$ **then**
      if $\mathtt{Ambiguity}(\mathtt{TransMonoid}(\mathscr{A}_{q_1+q_2}))$ **then**
        return IDA
      end if
    end if
  end if
end for
return Safe

Fig. 5: Algorithm of ambiguity analysis for regexes

The pseudocode of the complete algorithm[2] is given in Fig. 5. There $\mathscr{A}_{q_1+q_2}$ includes the orbit automata M_{q_1} and M_{q_2} of q_1 and q_2, and all states reachable from M_{q_1} and reaching M_{q_2} together with their transitions. Its initial state coincides with initial state of M_{q_1}, and its final states are final states of M_{q_2} (ignoring final states of $\mathscr{A}$ belonging either to M_{q_1} or intermediate states). The condition $c_1 \longrightarrow\!\!\!\!\!\rightarrow c_2$ ensures that the component c_2 is reachable from c_1, and they do not coincide. Operator $c[1]$ takes a first state from the component c (since the `Ambiguity.TransMonoid` and determinization tests results do not depend on the choice of the initial state in the orbit automata[3]). Function $\mathtt{SCC}(\mathscr{A})$ returns all strongly connected components of $\mathscr{A}$.

[2]The trial implementation of the method is given on https://github.com/bmstu-iu9/Chipo-Kleene/tree/ambiguity

[3]Absence of any useless states is guaranteed, because all the states are reachable from each other.

IV. Experiments

A. Data Set

In order to evaluate the effectiveness of our approach on the "domino" regexes, a dataset of 100 academic regexes was generated. The regexes satisfy the following properties:

- their length and alphabet are small (not more than 50 terms and not more than 5 distinct letters);
- they have iterated elements;
- all are in SSNF.

The first condition allows significant subexpression languages overlap, without blowing up the regex length. However, the test set contains not only complex dominoes, but also regexes with simple ambiguity situations like $\mathtt{b}^*\mathtt{c}(\mathtt{ac} \mid (\mathtt{aa} \mid \mathtt{a})^*\mathtt{d})^*$.

The second condition is necessary for REDoS situations. The third condition mostly excludes the trivial SSNF test, returning EDA value using our method too quickly.

We explored the dependence of the regexes matching time from the input length on the popular engines in Python, JavaScript, C++, Java 8, Java 11, Go, and Rust.

In order to detect super-linear dependencies, it is necessary to generate potentially attacking input, for which the string pumping method is used. The attacking input must match a pattern of 3 components: a *prefix* that satisfies the regular expression, a *pumping core* whose repetition can lead to a rapid increase in the number of parsing paths (i.e., malicious pump), and a *suffix* whose mismatch leads to catastrophic backtracking.

The results obtained by applying JavaScript, Python, C++ and Java 8 standard regex engines are the same, according to them, the data set contains 34 exponential, 36 polynomial and 30 safe regexes. Also, the experiments indicated that Java 11 standard regex engine handles some polynomial and exponential cases, but when the length of the input data increases significantly, it throws a stack overflow exception, which may be due to the introduction of the local storage of indexes to the regex module in the 11 version of Java. The regexes are safe for Go and Rust engines, which are based on the deterministic structures. Nevertheless, it was noted that there are frequent single outliers in trends when matching strings in Go.

During testing, we observed that polynomial regexes only lead to critical matching times (more than 1 minute) with significant input string lengths (approximately more than 500 characters), while expressions that have exponential matching complexity can reach critical time when parsing even relatively small input strings. In the simplest case, such a time explosion can be achieved with regexes that have large star nesting or multiple alternatives under a star quantifier. For instance, the Python, JavaScript, Java 8, and C++ regex engines are vulnerable to attacks in the case of the $((\mathtt{a}^*)^*)^*$ regex, and even the optimized Java 11 engine, which successfully handles double star nesting, reaches critical time processing such an expression.

However, more non-trivial cases were encountered in the proposed data set. For example, the regex

TABLE I: Time measurements

	Exponential		Polynomial		Safe		Unsafe		Timeouts
Tool	μ (s)	σ (s)	μ (s)	σ (s)	μ (s)	σ (s)	μ (s)	σ (s)	
RSA	1.895	2.614	3.480	3.748	0.836	0.341	2.578	3.221	13
ReScue	–	–	–	–	0.940	1.724	8.803	6.263	43
Revealer	0.410	0.035	0.402	0.021	0.320	0.065	0.409	0.033	0
Our method	0.846	1.059	1.178	1.259	0.484	0.400	1.014	1.169	0

TABLE II: Evaluation results

Tool	F_1-score	Total error rate	Vuln. error rate
RSA	0.90	0.13	0.00
ReScue	0.39	–	–
Revealer	0.55	0.45	0.04
Our method	1.00	0.00	0.00

`b(ab((a | b(a*a)*)a*b*)*|a*aaaa*)*`, when matched against the input of 32 characters that satisfies the pattern with prefix – `b`, pump – `abab`, suffix – `bbd`, achieves the following timings: PYTHON engine – over 3 minutes, JAVA 8 – over 3 minutes, JAVA 11 – 0.80 minutes, C++ – over 3 minutes, JAVASCRIPT – 1.73 minutes.

In general, the REDoS vulnerability degree coincides with the theoretical expectations, taking into account the asymptotic growth of the ambiguity function for the corresponding Thompson automata. Non-SSNF regexes cause critical time explosion, which is an evidence that the regex engines do not apply SSNF transformation to their input. In addition to non-SSNF regexes, critical REDoS situations occur on polynomial ambiguities iterated under a Kleene star.

B. Comparing with other in-research tools

We evaluated the effectiveness of the proposed approach by comparing it with three state-of-the-art open-source tools for detecting vulnerabilities in regexes: RSA [3], [17], a static analysis tool, RESCUE [5], [18], a genetic fuzzing tool, REVEALER [2], [19], an automated hybrid analysis tool that uses static and dynamic approaches.

The qualitative results of the experiments are described in Table II. To evaluate the effectiveness of detection of vulnerable and safe regexes, we used F_1-*score*, where *true positive* values are all vulnerable regular expressions that were classified as exponential or polynomial, the absence of results due to a timeout is taken into account as a false result, also we used the *error rate*, where a cumulative error on all classes of regexes – total error rate and a classification error among vulnerable regexes – vulnerable error rate. It should be noted that RESCUE does not support the exponential-polynomial classification, therefore, not all values were calculated for this tool.

The results of measuring the execution time for the considered tools are shown in Table I. When measuring time, all extended features of the tools were disabled, and their parameters were optimized. For each class of correctly classified regexes: exponential, polynomial, safe, unsafe (union of vulnerable regexes), the average running time (μ) and the standard deviation (σ) of this value were estimated, the amount of timeouts was also calculated.

Additionally, we chose 25 regexes with non-SSNF structure, which are analysed in our method by the preliminary ε-loop test. While our approach proved to be the fastest (which is not a surprise, provided the algorithm structure), the static part of REVEALER also had 100% success rate on this set, although, taking at average $4\times$ more time.

It is important to note that the theoretical results obtained by using static analysis methods, determining ambiguity degree of the Thompson automata, completely coincide with the experimental results obtained when testing the domino regexes on the PYTHON, JAVASCRIPT, JAVA 8, and C++ regex engines. This is a strong witness that regexes declared safe by dynamic or combined methods are their false negatives.

From the test results, we can conclude that the detection efficiency of the static analyser is high, but in non-trivial exponential or polynomial cases such as `(baa | ab)*b(a(b | a)ba*b)*(aab)*`, timeouts occur. The recognition efficiency of RESCUE and REVEALER tools on this data set is low. However, the proposed approach has the maximum quality of vulnerability detection, the average execution time is also superior to other implementations. This is partly explained by its narrow domain: testing only academic regexes. But RSA also aims at the academic regexes, and still has several timeouts; on the other hand, it seems that extension of REDoS-detection tools to non-academic regexes made them to miss *almost all* polynomial REDoS with domino structure.

V. DISCUSSION AND RELATED WORKS

Initially, our finite automata transforming tool was not designed to reveal REDoS situations. However, attempts to use open-source tools like Regex Static Analyser or RESCUE to analyze simple academic regexes with non-trivial ambiguity structure failed. The main purpose of the work was educational, so we designed our algorithm in such a way that it not only detects vulnerabilities, but also demonstrates them on the automata graphs (Fig. 6), at the cost of longer execution time. Since the tool was initially designed for demonstrations, only core academic regexes were considered. The algorithms used in the monoid-based approach have poor worst-case complexity, so its efficiency, compared to RSA and RESCUE, was a real surprise.

What features of the analysers caused such a situation? RSA uses NFA intersection construction, based on the well-known paper of Mohri et al [14]. To detect polynomial ambiguities, the algorithm requires self-intersecting an NFA

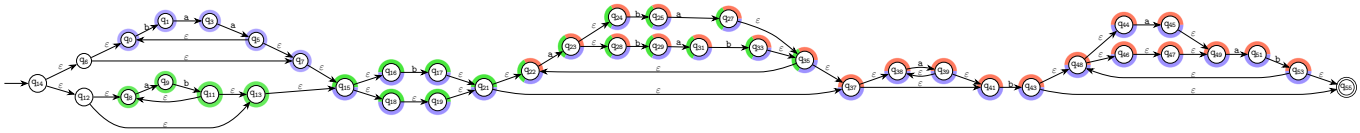


Fig. 6: Revealing strongly connected components with ambiguity situations in NFA graph

twice. The automata intersection problem is known to be PSPACE-complete [20], [21], thus, every additional intersection results in a significant slowdown. Maybe that is the main cause why the polynomial detection results in timeouts in RSA. The monoid and determinization algorithms are known to be worst-case exponential. However, the determinization is proven to be fast [4] in average [16], while the monoid representation depends heavily on the automata structure and, implemented to orbit automata, generates significantly fewer equivalence classes, compared to the case when automata are not cyclic. Another well-known problem in static analysers is dealing with ε-transitions, which can ruin the intersection construction, as well as the monoid. Surprisingly, the tools do not use the simple and natural conversion to the Glushkov construction preceded by the SSNF test.

Error rate of static tools is usually much lower than in tools using genetic algorithms and fuzzing, since REDoS-provoking strings can be disguised, requiring several explicit iterations to construct, or be combined from several alternative subexpressions under an iteration. Even using two approaches in REVEALER cannot help to find vulnerabilities, if the malicious pump is hidden in overlaps and crossing occurrences. For example, in paper [6], four REDoS classes are provided, based on a regex structure, and the regex `a*(ab)*a(ba)*` satisfies neither of them, because the vulnerability appears due to the crossing occurrence of the string `ab` on the border of the two orbits, whereas the expressions under Kleene stars have languages with empty intersection, which makes the regex "seemingly safe". A similar pattern-based approach is used in [7], resulting in the same sort of false negatives. So, regex-based heuristics showed themselves to be too weak as compared to the model NFA analysis in the domino ambiguity cases.

If a malicious pump for a regex is found, the natural question arises: how to correct the regex? We did not consider the whole implementation of the regex correction, but implemented a trial algorithm constructing a 1-unambiguous regex[5], if it exists [9]. However, for most regexes with overlaps, even if the equivalent 1-unambiguous regex can be built, the algorithm given in [9] produces exponentially longer result, as compared to the input, processing all overlap combinations separately. A more optimistic regex correcting heuristic is the Star Normal Form transformation: it is performed in linear time and produces regexes approximately of the same length. Moreover, the SSNF transformation is rather local, does not require transition to NFA, and can be applied even to extended regexes, which is useful, taking in account that non-SSNF regexes cause critical REDoS w.r.t. PYTHON and JAVASCRIPT regex engines. In general, the question what theoretical results can be used to fix REDoS regexes, is still a subject of research.

[4] And determinization-based Brzozowski minimisation algorithm frequently outperforms even Hopcroft's minimisation.

[5] 1-unambiguous regex is a regex whose Glushkov automaton is deterministic.

VI. CONCLUSION

The research resulted in the following answers to our research questions.

- *RQ1: how relevant is NFA static analysis w.r.t. to popular regex engines?*
  Our experiments demonstrated that the Thompson NFA model is entirely suitable for evaluating REDoS situations concerning the most widely used regex engines, including PYTHON, JAVASCRIPT, JAVA, and C++. Interestingly, although the GO regex machine uses conversion to DFA, it still produces surges on some ambiguous regexes with complex structures. The RUST DFA engine proved to be the most stable.
- *RQ2: what features of the REDoS analysers considered cause errors and time explosion on the regexes with complex overlap structure? How they can be processed reliably with less risk of time explosion?*
  We found out that considering orbit automata (instead of performing ambiguity analysis on the entire NFA) and using the Glushkov construction, preceded by the Strong Star Normal Form test, do not result in any loss of relevance, but significantly speed up the static analysis. Another interesting approach is to use monoid analysis as the primary ambiguity-detecting algorithm instead of NFA intersection analysis. If there are multiple substring overlaps in the orbits, this method performs significantly faster. However, if the overlaps are small, the number of equivalence classes in the monoid increases dramatically, making the intersection method more preferable.
  We also provided experimental evidence that the genetic search REDoS detection methods still miss complex REDoS cases, easily detected by static NFA analysis approaches.

Despite our approach proved itself to be efficient and reliable on the test set of domino regexes, it still requires many refinements. First, the monoid construction may explode if we take large alphabets, so the input regexes may need some alphabet factorization. E.g., if no overlaps are contained within a long string, then this string sometimes can be considered as a single letter. Second, it would be interesting to test the method on extended regexes approximation, and to

combine the monoid-based and intersection-based ambiguity detection algorithms.

REFERENCES

[1] J. C. Davis, C. A. Coghlan, F. Servant, and D. Lee, "The impact of regular expression denial of service (ReDoS) in practice: an empirical study at the ecosystem scale," in Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018, G. T. Leavens, A. Garcia, and C. S. Pasareanu, Eds. ACM, 2018, pp. 246–256. [Online]. Available: https://doi.org/10.1145/3236024.3236027

[2] Y. Liu, M. Zhang, and W. Meng, "Revealer: Detecting and exploiting regular expression denial-of-service vulnerabilities," in 2021 IEEE Symposium on Security and Privacy (SP), 2021, pp. 1468–1484.

[3] B. van der Merwe, N. Weideman, and M. Berglund, "Turning evil regexes harmless," in Proceedings of the South African Institute of Computer Scientists and Information Technologists, SAICSIT 2017, Thaba Nchu, South Africa, September 26-28, 2017, M. Masinde, Ed. ACM, 2017, pp. 38:1–38:10. [Online]. Available: https://doi.org/10.1145/3129416.3129440

[4] N. Weideman, B. van der Merwe, M. Berglund, and B. W. Watson, "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, ser. Lecture Notes in Computer Science, Y. Han and K. Salomaa, Eds., vol. 9705. Springer, 2016, pp. 322–334. [Online]. Available: https://doi.org/10.1007/978-3-319-40946-7_27

[5] Y. Shen, Y. Jiang, C. Xu, P. Yu, X. Ma, and J. Lu, "ReScue: Crafting regular expression DoS attacks," in Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ser. ASE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 225–235. [Online]. Available: https://doi.org/10.1145/3238147.3238159

[6] Y. Li, Y. Sun, Z. Xu, J. Cao, Y. Li, R. Li, H. Chen, S.-C. Cheung, Y. Liu, and Y. Xiao, "RegexScalpel: Regular expression denial of service (ReDoS) defense by Localize-and-Fix," in 31st USENIX Security Symposium (USENIX Security 22). Boston, MA: USENIX Association, Aug. 2022, pp. 4183–4200. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/li-yeting

[7] Y. Li, Z. Chen, J. Cao, Z. Xu, Q. Peng, H. Chen, L. Chen, and S. Cheung, "ReDoSHunter: A combined static and dynamic approach for regular expression DoS detection," in 30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021, M. Bailey and R. Greenstadt, Eds. USENIX Association, 2021, pp. 3847–3864. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/li-yeting

[8] Google. (2010–2023) Official public repository of RE2 library. [Online]. Available: https://github.com/google/re2

[9] A. Brüggemann-Klein and D. Wood, "One-unambiguous regular languages," Information and Computation, vol. 140, no. 2, pp. 229–253, 1998. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0890540197926882

[10] D. D. Freydenberger and M. L. Schmid, "Deterministic regular expressions with back-references," J. Comput. Syst. Sci., vol. 105, pp. 1–39, 2019. [Online]. Available: https://doi.org/10.1016/j.jcss.2019.04.001

[11] C. Allauzen and M. Mohri, "A unified construction of the Glushkov, Follow, and Antimirov automata," in Mathematical Foundations of Computer Science 2006, R. Královič and P. Urzyczyn, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 110–121.

[12] J. Éric Pin, "Mathematical foundations of automata theory," 2022.

[13] A. Weber and H. Seidl, "On the degree of ambiguity of finite automata," Theoretical Computer Science, vol. 88, no. 2, pp. 325–349, 1991. [Online]. Available: https://www.sciencedirect.com/science/article/pii/030439759190381B

[14] C. Allauzen, M. Mohri, and A. Rastogi, "General algorithms for testing the ambiguity of finite automata," in Developments in Language Theory, M. Ito and M. Toyama, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 108–120.

[15] A. Brüggemann-Klein, "Regular expressions into finite automata," Theor. Comput. Sci., vol. 120, no. 2, pp. 197–213, 1993. [Online]. Available: https://doi.org/10.1016/0304-3975(93)90287-4

[16] M. Almeida, N. Moreira, and R. Reis, "On the performance of automata minimization algorithms," 2007.

[17] (2015–2021) Regex static analyzer. [Online]. Available: https://github.com/NicolaasWeideman/RegexStaticAnalysis

[18] (2017–2023) Rescue. [Online]. Available: https://github.com/2bdenny/ReScue

[19] (2021) Revealer. [Online]. Available: https://github.com/cuhk-seclab/Revealer

[20] W. Gelade and F. Neven, "Succinctness of the Complement and Intersection of Regular Expressions," in 25th International Symposium on Theoretical Aspects of Computer Science, ser. Leibniz International Proceedings in Informatics (LIPIcs), S. Albers and P. Weil, Eds., vol. 1. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2008, pp. 325–336. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2008/1354

[21] J. Birget, S. W. Margolis, J. C. Meakin, and P. Weil, "Pspace-complete problems for subgroups of free groups and inverse finite automata," Theor. Comput. Sci., vol. 242, no. 1-2, pp. 247–281, 2000. [Online]. Available: https://doi.org/10.1016/S0304-3975(98)00225-4