Efficient Algorithms for Alternating Pushdown Systems with an Application to the Computation of Certificate Chains*

Dejvuth Suwimonteerabuth, Stefan Schwoon, and Javier Esparza

Institut für Formale Methoden der Informatik, Universität Stuttgart,
Universitätsstr. 38, 70569 Stuttgart, Germany
{suwimodh,schwoosn,esparza}@informatik.uni-stuttgart.de

Abstract. Motivated by recent applications of pushdown systems to computer security problems, we present an efficient algorithm for the reachability problem of alternating pushdown systems. Although the algorithm is exponential, a careful analysis reveals that the exponent is usually small in typical applications. We show that the algorithm can be used to compute winning regions in pushdown games. In a second contribution, we observe that the algorithm runs in polynomial time for a certain subproblem, and show that the computation of certificate chains with threshold certificates in the SPKI/SDSI authorization framework can be reduced to this subproblem. We present a detailed complexity analysis of the algorithm and its application, and report on experimental results obtained with a prototype implementation.

1 Introduction

Pushdown systems are a concept from formal-language theory that has turned out to be useful in computer-aided verification. They naturally model the behaviour of programs with possibly recursive procedures, and therefore model-checking for pushdown systems has been the subject of recent research. Burkhard and Steffen [1] and Walukiewicz [2] have studied the problem for the modal μ-calculus. Other papers [3–5] have investigated specialised algorithms for LTL model checking and both forward and backward reachability on pushdown systems. Concrete algorithms for these tasks with a precise complexity analysis were proposed in [5] and subsequently implemented in the Moped tool. Moreover, [3] has shown that a similar approach can be used to solve the *backward* reachability problem in alternating pushdown systems. This can be used to solve the model-checking problem for the alternation-free μ-calculus on (non-alternating) pushdown systems.

More recently, pushdown systems have also been applied in the field of computer security. In the authorization framework SPKI/SDSI [6], certificates are used to assign permissions to groups of principals, which are defined using local,

* This work was partially supported by the DFG project *Algorithms for Software Model Checking* and SFB 627 Nexus, Project A6.

hierarchical namespaces. In order to prove that a principal may access a certain resource, he/she needs to produce a chain of certificates that, taken together, provide a proof of authorisation. Jha and Reps [7] showed that a set of certificates can be seen as a pushdown system, and that certificate-chain discovery reduces to pushdown reachability. The SPKI/SDSI specification also provides for so-called *threshold certificates*, allowing specifications whereby a principal can be granted access to a resource if he/she can produce authorisations from multiple sources. We observe that this extension reduces to reachability on *alternating* pushdown systems.

Motivated by the applications in verification and authorisation, we study reachability algorithms for alternating pushdown systems (APDS) in more detail. The algorithm proposed in [3] is abstract (i.e. only the saturation rule is given), and its complexity is given as "exponential", without further details. Here, we provide a concrete algorithm for solving the problem together with a precise complexity analysis. Moreover, inspired by the work of [7], we show that the algorithm is very efficient for a special class of instances. Then, we consider two applications. The first one is straightforward: We show that the algorithm immediately leads to a procedure for computing winning regions in pushdown reachability games, and derive a complexity bound improving a previous analysis by [8]. The second application is perhaps more interesting. In [7], Jha and Reps observed that, for a restricted form of threshold certificates, the certificate-chain-discovery problem can be solved in polynomial, rather than exponential time. We prove this result again by showing that the existence of certificate chains can be reduced to the special class of instances of the reachability problem that we have identified. We perform a detailed complexity analysis, and report on a prototype implementation on top of the Nexus platform for context-aware systems [9].

We proceed as follows: Section 2 introduces alternating pushdown systems and other concepts used in the paper. Section 3 presents an algorithm for solving the reachability problem on APDS and analyzes its complexity. Section 4 studies the special class of instances mentioned above. Section 5 presents new upper bounds for computing winning regions in reachability pushdown games. Section 6 presents our application to certificate-chain discovery, and Section 7 reports experimental results.

2 Preliminaries

An *alternating pushdown system* (APDS) is a triplet $\mathcal{P} = (P, \Gamma, \Delta)$, where P is a finite set of *control locations*, Γ is a finite *stack alphabet*, and $\Delta \subseteq (P \times \Gamma) \times 2^{(P \times \Gamma^*)}$ is a set of *transition rules*. A *configuration* of $\mathcal{P}$ is a pair $\langle p, w \rangle$, where $p \in P$ is a control location and $w \in \Gamma^*$ is a *stack content*. If $((p, \gamma), \{(p_1, w_1), \ldots, (p_n, w_n)\}) \in \Delta$, we write $\langle p, \gamma \rangle \hookrightarrow \{\langle p_1, w_1 \rangle, \ldots, \langle p_n, w_n \rangle\}$ instead. We call a rule *alternating* if $n > 1$, or *non-alternating* otherwise. We also write $\langle p, \gamma \rangle \hookrightarrow \langle p_1, w_1 \rangle$ (braces omitted) for a non-alternating rule. Moreover, for every $w \in \Gamma^*$, the configuration $\langle p, \gamma w \rangle$ is an *immediate predecessor* of the set $\{\langle p_1, w_1 w \rangle, \ldots, \langle p_n, w_n w \rangle\}$.

A *computation tree* of $\mathcal{P}$ is a directed tree whose nodes are labelled by configurations and where every node n is either a leaf or an internal node labelled with c such that n has one outgoing hyperedge whose set of target nodes is labelled with configurations $C = \{c_1, \ldots, c_n\}$, where c is an immediate predecessor of C. We define the *reachability relation* $\Rightarrow$ as $c \Rightarrow C$ if there exists a computation tree such that c labels the root and C is the set of labels of the leaves. If $c \Rightarrow C$, then C is *reachable* from c. Given a set of configurations C, we define the set of *predecessors*, $pre^*(C) = \{c \mid \exists C' \subseteq C \colon c \Rightarrow C'\}$, as the set of configurations that are reachable backwards from subsets of C via the reachability relation.

Let us fix an APDS $\mathcal{P} = (P, \Gamma, \Delta)$. An alternating $\mathcal{P}$-automaton is a quintuple $\mathcal{A} = (Q, \Gamma, \delta, P, F)$, where $Q \supseteq P$ is a finite set of *states*, $F \subseteq Q$ is the set of *final states*, and $\delta \subseteq Q \times \Gamma \times 2^Q$ is a set of *transitions*. The *initial states* of $\mathcal{A}$ are the control locations of $\mathcal{P}$. We define the *transition relation* $\rightarrow \subseteq Q \times \Gamma^* \times 2^Q$ as the smallest relation satisfying:

- $q \xrightarrow{\varepsilon} \{q\}$ for every $q \in Q$,
- if $(q, \gamma, Q') \in \delta$ then $q \xrightarrow{\gamma} Q'$, and
- if $q \xrightarrow{w} \{q_1, \ldots, q_m\}$ and $q_i \xrightarrow{\gamma} Q_i$ for each $1 \leq i \leq m$, then $q \xrightarrow{w\gamma} (Q_1 \cup \ldots \cup Q_m)$.

$\mathcal{A}$ *accepts* or *recognizes* a configuration $\langle p, w\rangle$ if $p \xrightarrow{w} Q'$ for some $Q' \subseteq F$. The set of configurations recognized by $\mathcal{A}$ is denoted by $L(\mathcal{A})$.

In [3], it has been shown that given a set of configurations C of $\mathcal{P}$, recognized by an alternating automaton $\mathcal{A}$, we can construct another automaton $\mathcal{A}_{pre^*}$ such that $L(\mathcal{A}_{pre^*}) = pre^*(C)$.

The procedure of [3] assumes w.l.o.g. that $\mathcal{A}$ has no transition leading to an initial state. $\mathcal{A}_{pre^*}$ is computed by means of a saturation procedure, which adds new transitions to $\mathcal{A}$, according to the following rule:

> If $\langle p, \gamma\rangle \hookrightarrow \{\langle p_1, w_1\rangle, \ldots, \langle p_m, w_m\rangle\} \in \Delta$ and $p_1 \xrightarrow{w_1} P_1, \ldots, p_m \xrightarrow{w_m} P_m$ holds, then add $p \xrightarrow{\gamma} (P_1 \cup \ldots \cup P_m)$.

3 An Implementation for *pre**

In this section we present an implementation, as shown in Fig. 1, of the abstract algorithm from Sect. 2. Without loss of generality, the algorithm imposes two restrictions on every rule $\langle p, \gamma\rangle \hookrightarrow R$ in Δ:

(R1) if $R = \{\langle p', w'\rangle\}$, then $|w'| \leq 2$, and
(R2) if $|R| > 1$, then $|R| = 2$ and $\forall \langle p', w'\rangle \in R \colon |w'| = 1$.

Note that any APDS can be converted into an equivalent one that satisfies (R1) and (R2) with only a linear increase in size (i.e. the converted automaton executes the same sequences of actions, modulo the fact that one step may be refined into a sequence of steps).

In the rest of the paper we conduct a careful analysis in terms of certain parameters of the input, which are listed below:

- $\Delta_a, \Delta_0, \Delta_1, \Delta_2$ denote the sets of alternating rules and non-alternating rules with $0, 1, 2$ stack symbols in their right-hand side, respectively.
- The set of *pop control locations*, denoted by P_ε, is the set of control locations $p_1 \in P$ such that Δ_0 contains some rule $\langle p, \gamma \rangle \hookrightarrow \langle p_1, \varepsilon \rangle$.
- Given an alternating automaton, we define Q_{ni} as the set of its non-initial states, i.e., $Q_{ni} = Q \setminus P$.

Algorithm 1 computes $\mathcal{A}_{pre^*}$ by implementing the saturation rule. The sets *rel* and *trans* contain the transitions that are known to belong to $\mathcal{A}_{pre^*}$; *rel* contains those that have already been examined. Lines 1–4 initialize the algorithm. The rules $\langle p, \gamma \rangle \hookrightarrow \langle p_1, \epsilon \rangle$ are dealt with first, as in the *pre** algorithm of the non-alternating case [5]. All rules are copied to Δ' (line 3), and the auxiliary function $\mathcal{F}(r)$ is assigned to set of empty set for each rule r (line 4). The algorithm then proceeds by iteratively removing transitions from *trans* (line 6), adding them to *rel* if necessary (lines 7–8), and examining whether they generate other transitions via the saturation rule (lines 9–22). The idea of the algorithm is to avoid unnecessary operations. Imagine that the saturation rule allows to add transition t if transitions t_1 and t_2 are already present. Now, if t_1 is taken from *trans* but t_2 has not been added to $\mathcal{A}_{pre^*}$, we do not put t_1 back to *trans* but store the following information instead: if t_2 is added, then we can also add t. It turns out that these implications can be stored in the form of "fake pushdown rules" (like those added in line 18 or 21) and in the form of the auxiliary sets $\mathcal{F}(r)$.

Let us now look at the lines 9–22 in more detail. Lines 9–10 are as in [5]. Push rules (lines 11–19) and alternating rules (lines 20–22), however, require a more delicate treatment. At line 11 we know that $q \xrightarrow{\gamma} Q'$ is a transition of $\mathcal{A}_{pre^*}$ (because it has been popped from *trans*) and that $\langle p_1, \gamma_1 \rangle \hookrightarrow \langle q, \gamma\gamma_2 \rangle$ is a rule of the APDS. So we divide the states $q' \in Q'$ into those for which there is some rule $q' \xrightarrow{\gamma_2} Q''$ in *rel* and the rest. If there is no rest then we can add new rules to *trans* (lines 14–15). Otherwise we add the "fake rule" of line 18. At line 20 we know that $q \xrightarrow{\gamma} Q'$ is a transition of $\mathcal{A}_{pre^*}$ and $\langle p_1, \gamma_1 \rangle \hookrightarrow \{\langle q, \gamma \rangle\} \cup R$ is an alternating rule. So we add the "fake rule" $\langle p_1, \gamma_1 \rangle \hookrightarrow R$.

Note that the algorithm obviously runs with exponential time, since the number of transitions of A_{pre^*} can be exponential in the number of states. However, a closer look at the complexity reveals that the algorithm is exponential only in a proper subset of states, which can be small depending on the instance.

Lemma 1. *Algorithm 1 takes* $O(|\delta_0| + |\Delta_0| + |\Delta_1|2^n + (|\Delta_2|n + |\Delta_a|)4^n)$ *time, where* $n = |P_\varepsilon| + |Q_{ni}|$.

In typical applications, we start with a small automaton, i.e. δ_0 and Q_{ni} will be small. In that case, n will be dominated by $|P_\varepsilon|$, therefore the complexity can be simplified to $O(|\Delta_0| + |\Delta_1|2^{|P_\varepsilon|} + (|\Delta_2||P_\varepsilon| + |\Delta_a|)4^{|P_\varepsilon|})$

Theorem 1. *Let* $\mathcal{P} = (P, \Gamma, \Delta)$ *be an alternating pushdown system and* $\mathcal{A} = (Q, \Gamma, \delta_0, P, F)$ *be an alternating automaton. There exist an alternating automaton* $\mathcal{A}_{pre^*}$ *that recognizes pre*$^*(L(\mathcal{A}))$. *Moreover, if the restrictions R1 and R2*

Algorithm 1
Input: an APDS $\mathcal{P} = (P, \Gamma, \Delta)$;
an alternating $\mathcal{P}$-automaton $\mathcal{A} = (Q, \Gamma, \delta_0, P, F)$ without transitions into P
Output: the set of transitions of $\mathcal{A}_{pre^*}$

$\mathit{rel} := \emptyset$;
$\mathit{trans} := \delta_0 \cup \{\, (p, \gamma, p') \mid \langle p, \gamma \rangle \hookrightarrow \langle p', \varepsilon \rangle \in \Delta \,\} \cup \{\, (p, \gamma, \emptyset) \mid \langle p, \gamma \rangle \hookrightarrow \emptyset \in \Delta \,\}$;
$\Delta' := \Delta$;
$\mathcal{F} := \lambda x.\{\emptyset\}$;
while $\mathit{trans} \neq \emptyset$ **do**
  pop $t := (q, \gamma, Q')$ from trans;
  if $t \notin \mathit{rel}$ **then**
    add t to rel;
    for all $r := \langle p_1, \gamma_1 \rangle \hookrightarrow \langle q, \gamma \rangle \in \Delta'$ **and** $Q'' \in \mathcal{F}(r)$ **do**
      add $(p_1, \gamma_1, Q' \cup Q'')$ to trans;
    for all $\langle p_1, \gamma_1 \rangle \hookrightarrow \langle q, \gamma\gamma_2 \rangle \in \Delta'$ **do**
      $S := \{\, q' \in Q' \mid \exists Q'' : (q', \gamma_2, Q'') \in \mathit{rel} \,\}$;
      $\mathcal{Q}_1 := \{\, \bigcup_{q' \in S} Q_{q'} \mid \forall q' \in S \colon (q', \gamma_2, Q_{q'}) \in \mathit{rel} \,\}$;
      if $S = Q'$ **then**
        add $\{(p_1, \gamma_1, Q_1) \mid Q_1 \in \mathcal{Q}_1\}$ to trans;
      else
        $r := \langle p_1, \gamma_1 \rangle \hookrightarrow \{\langle q', \gamma_2 \rangle \mid q' \in Q' \setminus S\}$;
        add r to Δ';
        add $\mathcal{Q}_1$ to $\mathcal{F}(r)$;
    for all $r := \langle p_1, \gamma_1 \rangle \hookrightarrow \{\langle q, \gamma \rangle\} \cup R \in \Delta'$ s.t. $R \neq \emptyset$ **do**
      add $\langle p_1, \gamma_1 \rangle \hookrightarrow R$ to Δ';
      add $\{Q'' \cup Q' \mid Q'' \in \mathcal{F}(r)\}$ to $\mathcal{F}(\langle p_1, \gamma_1 \rangle \hookrightarrow R)$;
return rel;

Fig. 1. An algorithm for computing pre^*.

are met, $\mathcal{A}_{pre^*}$ can be constructed in $O(|\delta_0| + |\Delta_0| + |\Delta_1| 2^n + (|\Delta_2| n + |\Delta_a|) 4^n)$ time, where $n = |P_\varepsilon| + |Q_{ni}|$.

Given an APDS $\mathcal{P}$, a configuration c of $\mathcal{P}$, and a set of configurations C, the *backward reachability* problem for $\mathcal{P}$, c, and C is to check whether $c \in pre^*_{\mathcal{P}}(C)$. By Theorem 1, the problem is in EXPTIME. The following theorem shows a corresponding lower bound. It is a rather straightforward modification of a theorem of [10].

Theorem 2. *The backward reachability problem for alternating pushdown systems is **EXPTIME**-complete, even if C is a singleton.*

4 A Special Case

Recall the saturation rule of the abstract algorithm for the computation of pre^*: for every transition rule $\langle p, \gamma \rangle \hookrightarrow \{\langle p_1, w_1 \rangle, \ldots, \langle p_m, w_m \rangle\}$ and every set $p_1 \xrightarrow{w_1} P_1, \ldots, p_m \xrightarrow{w_m} P_m$, add a new transition $p \xrightarrow{\gamma} (P_1 \cup \ldots \cup P_m)$. The exponential complexity of the algorithm is due to the fact that the target of the new transition can be an arbitrary set of states, and so we may have to add an exponential number of new rules in the worst case. We now consider a special class of instances in which a new transition $p \xrightarrow{\gamma} Q$ need only be added if Q is a singleton, and show that a suitable modification of Algorithm 1 has polynomial running time.

Definition 1. *Let $\mathcal{P} = (P, \Gamma, \Delta)$ be an APDS, and let $R \subseteq P\Gamma^*$ be a set of configurations. We say that $(\mathcal{P}, R)$ is a* good *instance for the computation of pre* if for every $\langle p, d\rangle \hookrightarrow \{\langle p_1, w_1\rangle, \ldots, \langle p_n, w_n\rangle\} \in \Delta$ with $n \geq 2$ and for every $i \in \{1, \ldots, n\}$: $p_i w_i w \in pre^*(R)$ implies $w = \varepsilon$.*

I.e., if the set R can be reached from $p_i w_i$, then it cannot be reached from any $p_i w_i w$, where w is a nonempty word. As mentioned above, we introduce the following modification to the saturation rule: a new transition $p \xrightarrow{\gamma} Q$ is added only if Q is a singleton.

Theorem 3. *Let $\mathcal{P} = (P, \Gamma, \Delta)$ and R be a good instance, and let $\mathcal{A}$ be a nondeterministic automaton recognizing R. Assume w.l.o.g. that $\mathcal{A}$ has one single final state. Then, the modified saturation procedure produces a nondeterministic automaton recognizing the same language as $\mathcal{A}_{pre^*}$.*

Algorithm 1 implements the modified procedure after the following change to line 9: **for all** $r := \langle p_1, \gamma_1\rangle \hookrightarrow \langle q, \gamma\rangle \in \Delta'$ **and** $Q'' \in \mathcal{F}(r) \cap \{\emptyset, Q'\}$ **do**.

Lemma 2. *The modified Algorithm 1 takes $O(|\delta_0| + |\Delta_0| + (|\Delta_1| + |\Delta_a|)n + |\Delta_2|n^2)$ time, where $n = |P_\varepsilon| + |Q_{ni}|$, when applied to a good instance.*

Note that Algorithm 1, when applied to a *non-alternating* PDS (i.e. one with $\Delta_a = \emptyset$), has the same complexity as the algorithm from [5] that was specially designed for non-alternating PDS.

5 Computing Attractors in Pushdown Games

In [8] Cachat provided an algorithm for computing the winning positions of a player in a pushdown reachability game. It is straightforward to reformulate the algorithm in terms of *pre** computations for alternating pushdown automata. We do this, and apply the results of Sect. 3 to provide very precise upper bounds for the complexity of these problems.

A *pushdown game system* (PGS) is a tuple $\mathcal{G} = (P, \Gamma, \Delta_{\mathcal{G}}, P_0, P_1)$, where $(P, \Gamma, \Delta_{\mathcal{G}})$ is a PDS and P_0, P_1 is a partition of P. A PGS defines a pushdown game graph $G_{\mathcal{G}} = (V, \rightarrow)$ where $V = P\Gamma^*$ is the set of all configurations, and $p\gamma v \rightarrow qwv$ for every $v \in \Gamma^*$ iff $(p, \gamma, q, w) \in \Delta_{\mathcal{G}}$. P_0 and P_1 induce a partition $V_0 = P_0\Gamma^*$ and $V_1 = P_1\Gamma^*$ on V. Intuititively, V_0 and V_1 are the nodes at which players 0 and 1 choose a move, repectively. Given a start configuration $\pi_0 \in V$, a play is a maximal (possibly infinite) path $\pi_0\pi_1\pi_2 \ldots$ of $G_{\mathcal{G}}$; the transitions of the path are called *moves*; a move $\pi_i \to \pi_{i+1}$ is made by player 0 if $\pi_i \in V_0$; otherwise it is made by player 1.

The winning condition of a reachability game is a regular *goal set* of configurations $R \subseteq P\Gamma^*$. Player 0 wins those plays that visit some configuration of the goal set and also those that reach a deadlock for player 1. Player 1 wins the rest. We wish to compute the winning region for player 0, denoted by $Attr_0(R)$, i.e. the set of nodes from which player 0 can always force a visit to R or a deadlock for player 1. Formally [8]:

$$\begin{aligned} Attr_0^0(R) &= R \ , \\ Attr_0^{i+1}(R) &= Attr_0^i(R) \cup \{u \in V_0 \mid \exists v : u \to v, v \in Attr_0^i(R)\} \\ &\quad \cup \{u \in V_1 \mid \forall v : u \to v \Rightarrow v \in Attr_0^i(R)\} \ , \\ Attr_0(R) &= \bigcup\nolimits_{i \in \mathbb{N}} Attr_0^i(R) \ . \end{aligned}$$

Given a PGS $\mathcal{G} = (P, \Gamma, \Delta_{\mathcal{G}}, P_0, P_1)$, we define an APDS $\mathcal{P} = (P, \Gamma, \Delta)$ as follows. For every $p \in P$ and $\gamma \in \Gamma$: if $p \in P_0$, then for every rule $\langle p, \gamma\rangle \hookrightarrow \langle q, w\rangle$ of $\Delta_{\mathcal{G}}$ add the rule $\langle p, \gamma\rangle \hookrightarrow \{\langle q, w\rangle\}$ to Δ; if $p \in P_1$ and S is the set of right-hand-side configurations of rules with $\langle p, \gamma\rangle$ as left-hand-side, then add $\langle p, \gamma\rangle \hookrightarrow S$ to Δ. It follows immediately from the definitions that $Attr_0(R) = pre^*{}_{\mathcal{P}}(R)$ (intuitively, if $c \in pre^*{}_{\mathcal{P}}(R)$ then $c \Rightarrow C$ for some $C \subseteq R$, and so player 0 can force the play into the set C). So we can use Algorithm 1 to compute $Attr_0(R)$. To derive the complexity bound, we apply Lemma 1:

Theorem 4. *Let $\mathcal{G} = (P, \Gamma, \Delta_{\mathcal{G}}, P_0, P_1)$ be a PGS and a goal set R recognized by an alternating automaton $\mathcal{A}_R = (Q, \Gamma, \delta_0, P, F)$. An alternating automaton accepting the winning region can be computed in $O(|\delta_0| + |\Delta_0| + |\Delta_1| 2^n + (|\Delta_2| n + |\Delta_a|) 4^n)$ time, where $n = |P_\varepsilon| + |Q_{ni}|$.*

In [8] an upper bound of $O(|\Delta| \cdot 2^{c \cdot |Q|^2})$ is given. Our algorithm runs in $O(|\Delta| \cdot 2^{c \cdot |Q|})$ time, and in fact Theorem 4 further reduces the exponent $c \cdot |Q|$ to $|P_\varepsilon| + |Q_{ni}|$. Typically, $|P_\varepsilon| + |Q_{ni}|$ is much smaller than $|Q|$. First, recall that, because of the definition of $\mathcal{P}$-automaton, we have $P \subseteq Q$. Moreover, goal sets often take the form $p_1\Gamma^* \cup \ldots \cup p_n\Gamma^*$, i.e., player 0 wins if the play hits one of the control states $p_1, \ldots, p_n$. In this case we can construct $\mathcal{A}_R$ with $|Q_{ni}| = 1$. Since $|P_\varepsilon|$ is typically much smaller than $|P|$, the parameter n is much smaller than $|Q|$.

6 Computing Certificate Trees in SPKI/SDSI

In access control of shared resources, authorization systems allow to specify a security policy that assigns permissions to principals in the system. The *authorization problem* is, given a security policy, should a principal be allowed access to a specific resource? In frameworks such as SPKI/SDSI [6] and RT_0 [11], the security policy is expressed as a set of certificates, and the authorization problem reduces to discovering a subset of certificates proving that a given principal is allowed to access a given resource.

The SPKI/SDSI standard provides for so-called *threshold certificates*. Jha and Reps already observed in [7] that the authorization problem in the presence of such certificates can be reduced to the APDS reachability problem, and that a special case had polynomial complexity. In this paper, we observe that the special case corresponds to *good instances* of APDS reachability, as defined in Sect. 4, and provide a detailed complexity analysis. Moreover, we report on experimental results for a prototype implementation of the algorithm as an extension of the Nexus platform [9] with distributed access control.

We proceed in two steps. First, we consider "simple" SPKI/SDSI, a subset of SPKI/SDSI that has been considered in most of the work on this topic. Simple SPKI/SDSI does not handle threshold certificates, which we present in the second part. Finally, in Sect. 6.3, we discuss the application of our algorithms to RT_0.

6.1 Simple SPKI/SDSI

In this paper, we introduce only the basic notations that are required to understand SPKI/SDSI and its connections with alternating PDS. A more thorough explanation can be found in [7].

In SPKI/SDSI, the principals (individuals, resources, or any other entities) are represented by their public keys. We denote by $\mathcal{K}$ the set of public keys (or principals), specific keys are denoted by K, K_A, K', etc. An *identifier* is a word over some alphabet Σ (usually denoted by typewriter font such as $\mathtt{A}$, $\mathtt{B}$, $\ldots$). The set of identifiers is denoted by $\mathcal{A}$. A *local name* is of the form $K\ \mathtt{A}$, where $K \in \mathcal{K}$ and $\mathtt{A} \in \mathcal{A}$. For example, $K_X\ \mathtt{Customer}$ is a local name. A *term* is a key followed by zero or more identifiers. For example, $K\ \mathtt{Area\ Customer}$ is a term. SPKI/SDSI has two types of certificates, or "certs":

Name Certificates A name cert provides a definition of a local name in the issuer's local name space. Simply speaking, it can be understood as a rewrite rule of the form $K\ \mathtt{A} \rightarrow S$, where $K\ \mathtt{A}$ is a local name and and S is a term. Intuitively, this defines a meaning for $\mathtt{A}$ in the local name space of principal K, and only K may issue and sign such a cert.

Imagine, for instance, that X is a telecommunication company with multiple divisions, including the mobile phone division Xm. Alice is a customer with the mobile phone division. Consider the following certificates:

$$K_{Xm}\ \mathtt{customer} \rightarrow K_{Alice} \tag{1}$$

$$K_X\ \mathtt{customer} \rightarrow K_{Xm}\ \mathtt{customer} \tag{2}$$

Here, (1) intuitively declares Alice to be a customer of Xm, while (2) says that customers of Xm are also customers of the company X as a whole.

Authorization Certificates An auth cert grants or delegates a specific authorization from an issuer to a subject. It can be understood as a rewrite rule of the form $K_R \; \square \to S \; b$, where $b \in \{\square, \blacksquare\}$. If K_R is the owner of some resource R, then this certificate grants access to R to all principals described by term S. Only K_R may issue such a certificate. If $b = \square$, then authorized principals may delegate this authorization to other principals, otherwise delegation is not permitted. The following certificate grants access to resource R to all of X's customers, without delegation:

$$K_R \; \square \to K_X \; \texttt{customers} \; \blacksquare \tag{3}$$

Certificate Chains In order for Alice to prove that she has access to some resource, she needs to provide a list of certificates that lead from the public key to herself by applying left-prefix rewriting. Such a list of certificates is called a *certificate chain*. In the example, Alice is granted authorisation to access R if she can produce the certificate chain (3),(2),(1), because applying them (in this order) shows that:

$$K_R \; \square \overset{(3)}{\to} K_X \; \texttt{customers} \; \blacksquare \overset{(2)}{\to} K_{Xm} \; \texttt{customers} \; \blacksquare \overset{(1)}{\to} K_{Alice} \; \blacksquare$$

Since this chain leads from $K_R \; \square$ to $K_{Alice} \; \blacksquare$, Alice is authorised to access R, the "$\blacksquare$" indicating that she is unable to delegate that access further.

It was observed in [7] that a set of name and auth certs can be interpreted as a pushdown system; therefore, the authorization problem reduces to the problem of pushdown reachability and can be solved using the algorithms from [3, 5].

6.2 SPKI/SDSI with Threshold Certificates

The SPKI/SDSI standard [6] provides for so-called *threshold subjects*. A threshold subject is a pair $(\mathcal{S}, k)$ where $\mathcal{S}$ is a set of terms and $k \leq |\mathcal{S}|$. A threshold certificate is a name or auth cert where the right-hand side is a threshold subject. If threshold certificates are involved, proofs of authorisation can no longer be done purely by certificate chains. Instead, a proof of authorisation for Alice to access resource R becomes a *certificate tree*, where the nodes are labelled with terms and the edges are labelled with rewrite rules that can be applied to the term labelling their source nodes. The root is $K_R \; \square$, and if $K \; \texttt{A} \to (\mathcal{S}, k)$ is used to rewrite a node n, then the children of n are the elements of $\mathcal{S}$. The tree is considered a valid proof of authorisation for Alice if at least k of the children can be rewritten to $K_{Alice} \; b$, where $b \in \{\square, \blacksquare\}$.

We observe that it is sufficient to consider threshold certificates with subject $(\mathcal{S}, k)$ such that $k = |\mathcal{S}|$. (Any certificate where $k < |\mathcal{S}|$ can be simulated by $\binom{|\mathcal{S}|}{k}$ threshold certificates for each subset of $\mathcal{S}$ with exactly k elements.) Therefore, we will omit the number k from now on, silently assuming that it is equal to the cardinality of $\mathcal{S}$.

It can now easily be seen that in the presence of threshold certificates, the certificate set can be interpreted as an *alternating* pushdown system, and that the authorisation problem reduces to APDS reachability. In other words, Alice is granted access to resource R if she can prove that $K_R \,\square \Rightarrow \{K_{Alice} \,\square, K_{Alice} \,\blacksquare\}$.

In [12, 7] the use of threshold subjects is restricted to just authorization certificates, claiming that the use of threshold subjects in name certificates would make the semantics "almost surely too convoluted". Moreover, [7] observes that under this restriction the authorisation problem can be solved without incurring (asymptotic) run-time penalties for threshold subjects and gives an informal algorithm. Within our framework, we note that the restriction of threshold subjects to auth certs allows one to obtain a good instance and to apply the algorithm from Sect. 4 to solve the authorisation problem.

Theorem 5. *Let C_t, C_0, C_1, and C_2 be sets of certificates, where C_t contains the auth certs with threshold subjects, C_0 contains the name certs in which terms have zero identifiers, C_1 contains the name and auth certs in which terms have one and zero identifiers, respectively, and C_2 consists of the rest. Let n be the number of different terms in C_0. The authorization problem can be solved in $O(|C_0| + (|C_1| + |C_t|)n + |C_2|n^2)$ time.*

6.3 APDS reachability and RT_0

The RT_0 framework was proposed in [11], along with algorithms for solving its associated authorization problem. The expressiveness of RT_0 is very similar to that of SPKI/SDSI and allows for role intersection (similar to the threshold certificates of SKPI/SDSI). It is straightforward to assign a pushdown semantics to RT_0; therefore, our algorithms may serve as an alternative to the specialised algorithms in [11]. Let us regard this aspect in more detail.

The basic concepts in RT_0 are *entities* and *role names*, corresponding to "principals" and "identifiers" in SPKI/SDSI. A *role* is a tuple of the form $A.r$, where A is a principal and r is a role name, and corresponds to a local name in SPKI/SDSI. Certificates (or *credentials* in [11]) can be of the following types, where A, B are entities and r, r_1, r_2 are role names: (i) $A.r \leftarrow B$, (ii) $A.r \leftarrow B.r_1$, (iii) $A.r \leftarrow A.r_1.r_2$, and (iv) $A.r \leftarrow f_1 \cap \cdots \cap f_k$, for any $k \geq 1$, where $f_1, \ldots, f_k$ are roles.[1]

Unlike SPKI/SDSI, RT_0 does not distinguish between auth certs and name certs; in fact, all certs in RT_0 can be understood as name certs, including those that deal with role intersection. The semantics of a set of certs, then, is a mapping *Mem* of roles to the sets of entities, i.e. the least fixpoint satisfying the following equations:

(i) if $A.r \leftarrow B$ is a cert, then $B \in \mathit{Mem}(A.r)$;
(ii) if $A.r \leftarrow B.r_1$ is a cert, then $\mathit{Mem}(B.r_1) \subseteq \mathit{Mem}(A.r)$;
(iii) if $A.r \leftarrow A.r_1.r_2$ is a cert, then $\bigcup_{B \in \mathit{Mem}(A.r_1)} \mathit{Mem}(B.r_2) \subseteq \mathit{Mem}(A.r)$;
(iv) if $A.r \leftarrow f_1 \cap \cdots \cap f_k$ is a cert, then $\bigcap_{i=1}^{n} \mathit{Mem}(f_i) \subseteq \mathit{Mem}(A.r)$.

[1] In fact, [11] permits $f_1, \ldots, f_k$ to be any of the expressions allows on the right-hand side of rules (i), (ii), or (iii); however, our restriction allows for an easier presentation of the semantics without restricting the expressiveness of the framework.

The *membership problem* in RT_0 is to determine, given a role $A.r$ and an entity B, whether $B \in \mathit{Mem}(A.r)$. RT_0 does not provide for auth certs; the underlying assumption is that permissions can be assigned to roles by external means.

It can easily be seen that rules of type (i), (ii), and (iii) can be given a (non-alternating) pushdown semantics such that the resulting PDS satisfies $\langle A, r\rangle \Rightarrow \{\langle B, \varepsilon\rangle\}$ if and only if $B \in \mathit{Mem}(A.r)$. Thus, the membership problem for RT_0 without certs of type (iv) can be solved using algorithms for (non-alternating) pushdown reachability [5].[2]

A straightforward translation of rules of type (iv) to *alternating* pushdown rules fails, however. It turns out that the semantics of type-(iv) rules is subtly restrictive, making RT_0 less expressive than APDS. For instance, consider the following set of RT_0 credentials:

$$A.a \leftarrow A.b.c, \quad A.b \leftarrow B.d \cap C.e, \quad B.d \leftarrow D, \quad C.e \leftarrow E, \quad D.c \leftarrow F, \quad E.c \leftarrow F$$

We have $\mathit{Mem}(B.d) = \{D\}$ and $\mathit{Mem}(C.e) = \{E\}$, therefore $\mathit{Mem}(A.b) = \emptyset$, and consequently $\mathit{Mem}(A.a) = \emptyset$. It is tempting to translate the rule $A.b \leftarrow B.d \cap C.e$ to the alternating pushdown rule $\langle A.b\rangle \hookrightarrow \{\langle B.d\rangle, \langle C.e\rangle\}$. But then, the resulting APDS satisfies $\langle A.a\rangle \Rightarrow \{\langle F, \varepsilon\rangle\}$, which is at odds with the RT_0 semantics $\mathit{Mem}(A.a) = \emptyset$.

However, the following relationship holds: Let $\mathcal{C}$ be a set of RT_0 credentials, and let $\mathcal{P}$ be the APDS derived from $\mathcal{C}$ in the manner described above. Let $R = \{\langle B, \varepsilon\rangle\}$ be a singleton set of configurations for some entity B. Then, for any role $A.r$, it holds that $B \in \mathit{Mem}(A.r)$ if and only if $\langle A, r\rangle \Rightarrow R$ *provided that* $(\mathcal{P}, R)$ *is a good instance.* Notice that the APDS resulting from the previous example and the set $\{\langle F, \varepsilon\rangle\}$ do not form a good instance because the APDS contains a rule $\langle A, b\rangle \hookrightarrow \{\langle B, d\rangle, \langle C, e\rangle\}$, but $\langle B, dc\rangle$ and $\langle C, ec\rangle$ are both contained in $\mathit{pre}^*(\{\langle F, \varepsilon\rangle\})$.

Previous algorithms for solving membership queries were proposed in [11] and [13]. The algorithms in [11] are based on the concept of so-called credential graphs, whereas [13] employs logic programs. However, the best known time bounds for the membership problem given in [11, 13] are cubic in the number of credentials. The algorithm provided in Sect. 4 can solve the membership problem with only linear time in the number of credentials and quadratic only in the number of entities that are the target of a type (i) credential.

7 Implementation and Experiments

We have implemented a prototype of the pre^* algorithm for APDS (in fact, a dedicated version for good instances) inside the Nexus platform [9]. An application can use Nexus "middleware" in order to obtain context data about mobile objects registered at the platform, like the position of an object or whether it enjoys a given relation to another object.

[2] This fact was already mentioned in [13].

Nexus is based on an *Augmented World Model* (AWM). AWM can contain both real world objects (e.g. rooms or streets) and virtual objects (e.g. websites). Furthermore, Nexus defines a language called *Augmented World Modeling Language* (AWML). This XML-based language is used for exchanging Nexus objects between the platform and data repositories.

Our prototype extends the AWM and AWML with name and authorization relations, which can be viewed as name and authorization certificates in the case of SPKI/SDSI, respectively. In other words, we model relations as virtual objects in the Nexus context. Moreover, we extend the platform so that it can serve applications querying relations between entities. Note that, normally, the base information about objects is contained in a Nexus database (the so-called context server) and returned in the form of AWML documents. Our prototype is not yet connected to such a database; instead, all data is kept directly in AWML.

7.1 A Scenario

Consider a scenario where company X takes part in a trade fair. The exhibition center consists of 2 exhibitions. An exhibition's area is a hierarchical structure with 3 exhibition halls, divided into 4 floors with 5 booths each. The structure can be written by pushdown rules as follows, given that $1 \leq i \leq 2, 1 \leq j \leq 3, 1 \leq k \leq 4, 1 \leq l \leq 5$:

$$E_i \; \texttt{Area} \rightarrow E_i \; \texttt{Hall Floor Booth} \tag{4}$$
$$E_i \; \texttt{Hall} \rightarrow H_{[i,j]} \tag{5}$$
$$H_{[i,j]} \; \texttt{Floor} \rightarrow F_{[i,j,k]} \tag{6}$$
$$F_{[i,j,k]} \; \texttt{Booth} \rightarrow B_{[i,j,k,l]} \tag{7}$$

Now, company X launches a promotion for visitors of the exhibition center to freely download ringtones for their mobile phones. The following visitors are allowed to download: (1) customers of X who are currently in the area of exhibition 1; (2) non-customers to whom the right has been delegated by one of X's customers; (3) customers who are currently not in the area of exhibition 1, but have received delegation from another visitor of exhibition 1. This is expressed by the following rule:

$$K_X \; \square \rightarrow \{E_1 \; \texttt{Area Visitor} \; \square, K_X \; \texttt{Customer} \; \square\} \tag{8}$$

The facts that Alice is visiting a booth in exhibition 1, and that she delegates her right to Bob, who is a customer of X, can be written as:

$$B_{[1,j,k,l]} \; \texttt{Visitor} \rightarrow K_{Alice}, \qquad \text{for some } j, k, l \tag{9}$$
$$K_{Alice} \; \square \rightarrow K_{Bob} \; \blacksquare \tag{10}$$
$$K_X \; \texttt{customer} \rightarrow K_{Bob} \tag{11}$$

When Bob wants to download a ringtone, we can efficiently compute the set $pre^*(\{\langle K_{Bob}, \square\rangle, \langle K_{Bob}, \blacksquare\rangle\})$ by noting the fact that the rules (4)–(11) and

$\{\langle K_{Bob}, \square\rangle, \langle K_{Bob}, \blacksquare\rangle\}$ form a good instance. Bob's request is granted in this case because $\langle X, \square\rangle \in \mathit{pre}^*(\{\langle K_{Bob}, \square\rangle, \langle K_{Bob}, \blacksquare\rangle\})$. Note that Bob can only download as long as Alice stays in booths in the exhibition 1. As soon as she moves away (i.e. the rule (9) is removed), a request from Bob can no longer be granted even though he is a customer of X.

7.2 Experiments

The scenario explained above is implemented as an application of the Nexus platform. We report on the running time for some experiments. The experiments should give a rough idea of the size of problems that can be handled in reasonable time.

We randomly add visitors to the exhibition center, and let them randomly issue certificates. We consider a base case with 1000 visitors in the exhibition center, 100 of them are customers of the company X, and the visitors issue 1000 authorization certificates. The issuer of a certificate decides randomly whether the right can be further delegated or not. The series were conducted on a 2 GHz PC with 256 MB RAM.

7.3 Experiment 1

In the base case, 10 % of visitors are customers of X, and a visitor issues one certificate on average. In our first experiment we keep these two ratios constant, and increase the number of visitors (for example, if there are 2000 visitors, there will be 200 customers that authorize 2000 times). We ran the experiment five times for each set of parameters. In each run 1000 random download requests are made. Table 1 displays the average results for 1000, 2000, 5000, and 10000 visitors (V). The table shows how often the request was granted (G) and rejected (R), the average time of a certificate search (T), and average time for granted (T(G)) and rejected (T(R)) searches. All measurements are in milliseconds.

In a realistic scenario, solving the authorisation problem requires to query databases (e.g. databases containing the positions of objects) and transmit data over a network, which are comparatively expensive operations. We kept relations of various types in different AWML files and whenever a piece of data was needed, we retrieved it from there. Since opening and reading files is also a comparatively expensive operation, this gives some insight as to the overhead such operations would incur in practice. The table shows the number of times AWML files (F) needed to be opened in average. For comparison, the numbers for granted (F(G)) and rejected (F(R)) requests are also displayed.

This experiment allows to draw a first conclusion: The average time of a search does not depend on the number of visitors per se. When a visitor requests a download, the algorithm has to search for the issuers of its certificates. Since the number of certificates is equal to the number of visitors, each visitor has one certificate in average.

Table 1. Results of Experiment 1

V	G	R	T	T(G)	T(R)	F	F(G)	F(R)
1000	229.8	770.2	18.71	29.09	15.49	13.84	22.54	11.19
2000	195.6	804.4	19.23	28.76	16.92	13.14	21.25	11.16
5000	202.2	797.8	18.62	29.33	15.90	12.99	21.10	10.93
10000	199.4	800.6	24.90	38.25	21.60	13.00	22.00	10.77

7.4 Experiment 2

In this experiment, we kept the number of visitors constant, and increased the number of certificates they issue, shown in column C in Table 2. The other columns are as in Experiment 1. Again, we ran the experiment five times for each value of C. Each run consisted of 100 random requests.

Table 2. Results of Experiment 2.

C	G	R	T	T(G)	T(R)	F	F(G)	F(R)
1000	23.0	77.0	18.71	29.09	15.49	13.84	22.54	11.19
2000	56.2	43.8	120.72	193.93	21.96	74.68	118.50	15.83
3000	86.4	13.6	1477.35	1704.21	33.66	625.41	721.69	12.91
4000	95.2	4.8	2279.13	2393.81	13.40	898.01	942.94	9.64

We see that the running time grows rapidly with the number of certificates issued. The explanation is the larger number of certificates received by each visitor, which leads to many more certificate chains. Observe also that the number of granted requests increases.

The overall conclusion of the two experiments is that the algorithm scales well to realistic numbers of visitors and certificates. Notice that in the intended application a user will be willing to wait for a few seconds.

8 Conclusions

We have provided an efficient implementation of the saturation algorithm of [3] for the computation of pre^* in alternating pushdown systems. Following [8], we have applied the algorithm to the problem of determining the winning region in reachability pushdown games, improving the complexity bound of [8]. We have shown that the algorithm has very low complexity for certain good instances, and provided an application: The computation of certificate chains with threshold subjects in the SPKI/SDSI authorization framework can be reduced to these instances. We have implemented the algorithm within the Nexus platform [9], and shown that it scales up to realistic scenarios.

References

1. Burkart, O., Steffen, B.: Model checking the full modal mu-calculus for infinite sequential processes. In: Proc. ICALP. LNCS 1256, Springer (1997) 419–429
2. Walukiewicz, I.: Pushdown processes: Games and model checking. In: Proc. CAV. LNCS 1102 (1996) 62–74
3. Bouajjani, A., Esparza, J., Maler, O.: Reachability analysis of pushdown automata: Application to model-checking. In: Proc. CONCUR. LNCS 1243 (1997) 135–150
4. Finkel, A., Willems, B., Wolper, P.: A direct symbolic approach to model checking pushdown systems. ENTCS **9** (1997)
5. Esparza, J., Hansel, D., Rossmanith, P., Schwoon, S.: Efficient algorithms for model checking pushdown systems. In: Proc. CAV. LNCS 1855 (2000) 232–247
6. Ellison, C., Frantz, B., Lampson, B., Rivest, R., Thomas, B., Ylönen, T.: RFC 2693: SPKI Certificate Theory. The Internet Society. (1999)
7. Jha, S., Reps, T.: Model checking SPKI/SDSI. JCS **12**(3–4) (2004) 317–353
8. Cachat, T.: Symbolic strategy synthesis for games on pushdown graphs. In: Proc. ICALP. LNCS 2380 (2002) 704–715
9. Hohl, F., Kubach, U., Leonhardi, A., Rothermel, K., Schwehm, M.: Nexus - an open global infrastructure for spatial-aware applications. Technical Report 1999/02, Universität Stuttgart: SFB 627 (1999)
10. Chandra, A., Kozen, D., Stockmeyer, L.: Alternation. JACM **28**(1) (1981) 114–133
11. Li, N., Winsborough, W.H., Mitchell, J.C.: Distributed credential chain discovery in trust management. Journal of Computer Security **11**(1) (2003) 35–86
12. Clarke, D., Elien, J., Ellison, C., Fredette, M., Morcos, A., Rivest, R.: Certificate chain discovery in SPKI/SDSI. At `http://theory.lcs.mit.edu/~rivest/` (1999)
13. Li, N., Mitchell, J.C., Winsborough, W.H.: Beyond proof-of-compliance: Security analysis in trust management. Journal of the ACM **52**(3) (2005) 474–514

Appendix

We now show that Algorithm 1 correctly implements the saturation procedure from [3], and analyse its complexity. The results are summarized in Theorem 1, however we need the following three lemmata:

Lemma A1. *Algorithm 1 terminates.*

Proof. Since *rel* is initially empty and Q, Γ are finite sets, *rel* is a finite set. Therefore, the block at lines 8–22 can only be executed finitely many times. Δ' is finite, since Δ is finite and only finitely many rules are added to Δ'. Hence, all loops after line 8 terminate, and only finite number of elements can be added to *trans*. Once *rel* can no longer grow, *trans* can no longer grow and will be empty eventually. This causes the algorithm to terminate. □

Lemma A2. *Upon termination of Algorithm 1, rel is equal to the set δ of transitions of $\mathcal{A}_{pre^*}$.*

Proof. We divide the proof into two parts:

"$\subseteq$": We show that throughout the algorithm *rel* $\subseteq \delta$ holds. In Sect. 2, δ was defined to contain δ_0 and satisfy the saturation rule. *rel* contains only elements from *trans*, so we inspect the lines that change *trans*, and show that all additions to *trans* satisfy saturation rules:

- Line 2: *trans* is initialized to δ_0, and also the saturation rule is directly modelled here.
- Line 15: models the saturation rule in the case of $\langle p_1, \gamma_1\rangle \hookrightarrow \langle q, \gamma\gamma_2\rangle \in \Delta'$ (line 11), $(q, \gamma, Q') \in$ *trans* (line 6), and $S = Q'$ (line 14), where S is the set of control locations q' from Q' such that (q', γ_2, Q'') was in *trans* for some Q'' (line 12). At line 13 $\mathcal{Q}_1$ keeps unions of Q'' for each different element from S. When $S = Q'$, it means that every element in Q' has outgoing transitions labelled with γ_2. Line 15 add (p_1, γ_1, Q_1) to *trans* for every $Q_1 \in \mathcal{Q}_1$. Thus, it models the saturation rule.
- Line 10: The rule $\langle p_1, \gamma_1\rangle \hookrightarrow \langle q, \gamma\rangle$ was added to Δ' because of either of the following three reasons:
  1. The rule was added to Δ' at line 3, i.e. $\langle p_1, \gamma_1\rangle \hookrightarrow \langle q, \gamma\rangle \in \Delta$ and $\mathcal{F}(\langle p_1, \gamma_1\rangle \hookrightarrow \langle q, \gamma\rangle) = \emptyset$. The saturation rule directly applies.
  2. The rule was added to Δ' at line 18, which implies that r at line 17 was $\langle p_1, \gamma_1\rangle \hookrightarrow \langle q, \gamma\rangle$, and the loop at line 11 was entered with some p'', γ', and Q' such that $\langle p_1, \gamma_1\rangle \hookrightarrow \langle p'', \gamma'\gamma\rangle \in \Delta$ and $(p'', \gamma', Q') \in$ *trans*. Since $r := \langle p_1, \gamma_1\rangle \hookrightarrow \langle q, \gamma\rangle$, we know that $S = Q' \setminus \{q\}$ at line 12, and $\mathcal{Q}_1$ gets unions of Q'' such that (q', γ, Q'') was in *trans* for each different $q' \in S$ at line 13. $\mathcal{Q}_1$ was added to $\mathcal{F}(r)$ at line 19. Then, when r is considered at line 10 because of the transition (q, γ, Q'), we add transitions $(p_1, \gamma_1, Q' \cup Q'')$ for each $Q'' \in \mathcal{F}(r)$ according to the saturation rule.
  3. The rule was added to Δ' at line 21, which implies that the loop at line 20 was entered with some q_1', γ_1', and Q_1' such that $r := \langle p_1, \gamma_1\rangle \hookrightarrow \{\langle q_1', \gamma_1'\rangle\} \cup \{\langle q, \gamma\rangle\} \in \Delta'$ (i.e. $R = \{\langle q, \gamma\rangle\}$) and the transition $(q_1', \gamma_1', Q_1') \in$ *trans*. The states Q_1' was saved by adding to $\mathcal{F}(\langle p_1, \gamma_1\rangle \hookrightarrow \langle q, \gamma\rangle)$ unions of Q_1' and Q'' for each $Q'' \in \mathcal{F}(r)$ at line 22. Now, we consider again where the rule r was added to Δ'. There are three cases:
     - 3.1 If r was added to Δ' at line 3, i.e. $r \in \Delta$, adding transitions $(p_1, \gamma_1, Q' \cup Q'')$ for each $Q'' \in \mathcal{F}(\langle p_1, \gamma_1\rangle \hookrightarrow \langle q, \gamma\rangle)$ therefore conforms to the saturation rule.
     - 3.2 If r was added to Δ' at line 18, then r at line 17 was $\langle p_1, \gamma_1\rangle \hookrightarrow \{\langle q, \gamma\rangle, \langle q_1', \gamma_1'\rangle\}$, and the loop at line 11 must be entered with some p'', γ', and Q' such that $\langle p_1, \gamma_1\rangle \hookrightarrow \langle p'', \gamma'\gamma\rangle \in \Delta$ and $(p'', \gamma', Q') \in$ *trans*. Since $r := \langle p_1, \gamma_1\rangle \hookrightarrow \{\langle q, \gamma\rangle, \langle q_1', \gamma_1'\rangle\}$ at line 17, we know that $\gamma = \gamma'$, $S = Q' \setminus \{q, q_1'\}$ at line 12, and $\mathcal{Q}_1$ gets unions of Q'' such that (q', γ, Q'') was in *trans* for each different $q' \in S$ at line 13. $\mathcal{Q}_1$ was added to $\mathcal{F}(r)$ at line 19. Again, when considering $\langle p_1, \gamma_1\rangle \hookrightarrow \langle q, \gamma\rangle$, we add $(p_1, \gamma_1, Q' \cup Q'')$ for each $Q'' \in \mathcal{F}(\langle p_1, \gamma_1\rangle \hookrightarrow \langle q, \gamma\rangle)$ according to the saturation rule.

3.3 If r was added to Δ' at line 21, then there must eventually be $\langle p_1, \gamma_1 \rangle \hookrightarrow \{\langle q, \gamma \rangle, \langle q_1', \gamma_1' \rangle, \ldots, \langle q_n', \gamma_n' \rangle\} \in \Delta'$ for some $n \geq 2$ such that this rule was added at line 18. The reason is because of the restriction (R2), which means only line 18 can add such a rule to Δ'. We can perform the analysis similar to 3 and 3.2, and conclude that the addition conforms to the saturation rule.

"$\supseteq$": We show that upon termination $rel \supseteq \delta$ holds. Equivalently, we prove that by the time the algorithm terminates, all possible saturation rules have been applied. Two cases are considered:

Case 1: Assume $\langle p, \gamma \rangle \hookrightarrow \langle p', w \rangle \in \Delta$ and there is $p' \xrightarrow{w} Q$ in rel.

* If $w = \varepsilon$, then $Q = \{p'\}$ and (p, γ, p') has been added in line 2.
* If $w = \gamma_1$ and $(p', \gamma_1, Q) \in rel$, then (p, γ, Q) has been added in line 10.
* If $w = \gamma_1\gamma_2$, and rel contains transitions $t' = (p', \gamma_1, Q')$ and $t_j'' = (q_j', \gamma_2, Q_j')$ for every $q_j' \in Q'$, at line 12 S contains control locations q_j' such that t_j'' was already examined, and unions of Q_j' for each different $q_j' \in S$ are saved in $\mathcal{Q}_1$ at line 13.
  · If t' was examined after all t_j'', then $S = Q'$, which means $(p, \gamma, \bigcup_{j, q_j \in S} Q_j')$ was added in line 15.
  · If t' was examined before t_j'' for some j, Δ' has the rule $r := \langle p, \gamma \rangle \hookrightarrow \{\langle q_j', \gamma_2 \rangle \mid q_j' \in Q' \setminus S\}$ at line 18, and $\mathcal{Q}_1$ was added to $\mathcal{F}(r)$ at line 19. When t_k'', where $q_k' \notin S$, was later examined, there are two possible cases depending on the number of elements in the right-hand-side set of the rule.
  1. If there is more than one element in the set, line 21 adds a rule without $\langle q_k'', \gamma_2 \rangle$ in the right-hand-side set to Δ', and Q_k' was added to $\mathcal{F}$ of this new rule at line 22. This step is repeated for each different t_k'' until there is one element in the right-hand-side set.
  2. If there is one element in the set, then $(p, \gamma, Q' \cup Q'')$ is added for each $Q'' \in \mathcal{F}(r)$ at line 10.

Case 2: Assume $\langle p, \gamma \rangle \hookrightarrow \{\langle q_1, \gamma_1 \rangle, \langle q_2, \gamma_2 \rangle\} \in \Delta$, rel contains $t_1 = (q_1, \gamma_1, Q_1)$, and $t_2 = (q_2, \gamma_2, Q_2)$.

* If t_1 was examined before t_2, Δ' has the rule $r = \langle p, \gamma \rangle \hookrightarrow \langle q_2, \gamma_2 \rangle$ at line 21, and $\mathcal{F}(r)$ contains Q_1 at line 22. When t_2 was examined, $(p, \gamma, Q_1 \cup Q_2)$ was added in line 10.
* If t_2 was examined before t_1, the proof is analogous.

□

Lemma 1. *Algorithm 1 takes* $O(|\delta_0| + |\Delta_0| + |\Delta_1|2^n + (|\Delta_2|n + |\Delta_a|)4^n)$ *time, where* $n = |P_\varepsilon| + |Q_{ni}|$.

Proof. Let ν be the smallest set of states such that for every transition (q, γ, Q') that is added to *trans* at any any point during the algorithm, Q' only contains states of ν. Because of line 2 of the algorithm, we have $P_\varepsilon \cup Q_{ni} \subseteq \nu$. However, the other lines that add transitions to *trans* (namely, 10 and 15) do not add more elements to ν, since every transition (q, γ, Q') they add, Q' must be a union of Q'' from some (q', γ', Q'') that were in *trans*. So, $\nu = P_\varepsilon \cup Q_{ni}$.

Because of the definition of ν, the number of sets $Q' \subseteq Q$ for which after termination of the algorithm *rel* contains a transition of the form (q, γ, Q') is $2^{|\nu|} = 2^n$.

We now consider the number of times the statements inside the main loop are executed. Line 12 is executed once for each combination of $\langle p_1, \gamma_1 \rangle \hookrightarrow \langle q, \gamma\gamma_2 \rangle \in \Delta_2$ and (q, γ, Q'), i.e. $O(|\Delta_2|2^n)$ times. Thus, line 18 is executed $O(|\Delta_2|2^n)$ times. The alternating rules inside Δ' only come from Δ_a and line 18, hence there are $O(|\Delta_a| + |\Delta_2|2^n)$ of them. Line 21 is executed once for each combination of alternating rule $\langle p_1, \gamma_1 \rangle \hookrightarrow R \cup \{\langle q, \gamma \rangle\} \in \Delta'$ and (q, γ, Q'). Therefore, line 21 is executed $O(|\Delta_a|2^n + |\Delta_2|4^n)$ times. Moreover, since Q' and γ_2 are fixed, line 15 is executed $O(|\Delta_2|4^n)$ times.

Line 10 is executed once for each combination of non-alternating rule in Δ' and (q, γ, Q'). Since the size of Δ' of this form is $O(|\Delta_1| + |\Delta_2|n + |\Delta_a|)$ and $f(r)$ has at most 2^n elements for each rule $r \notin \Delta_1$, line 10 is executed $O(|\Delta_1|2^n + (|\Delta_2|n + |\Delta_a|)4^n)$ times.

We now count the iterations of the main loop, i.e. how often line 6 is executed. This directly depends on the number of elements added to *trans*. Initially, *trans* has $|\delta_0| + |\Delta_0|$ elements from lines 1 and 2. The time complexity can then be concluded from the fact that all addtions to *trans* are no more than $O(|\delta_0| + |\Delta_0| + |\Delta_1|2^n + (|\Delta_2|n + |\Delta_a|)4^n)$ in number. $\square$

Theorem 2. *The backward reachability problem for alternating pushdown systems is **EXPTIME**-complete, even if C is a singleton.*

Proof. We use a reduction from the acceptance problem for alternating Turing machines.

More specifically, let $\mathcal{M} = (Q, \Sigma, \delta, q_0)$ be an alternating Turing machine, where the control states Q are partitioned into *existential*, *universal*, *accepting*, and *rejecting* states, and where $\delta \colon Q \times \Sigma \to 2^{Q \times \Sigma \times \{L,N,R\}}$ is the transition function.

Let us assume that, when started on the input w, $\mathcal{M}$ uses at most $p(|w|)$ space on the tape, where p is some polynomial independent of w. Thus, a configuration of $\mathcal{M}$ can be represented by a word from $\Sigma^* Q \Sigma \Sigma^*$ of length $p(|w|)$. In a configuration uqv, u is are the tape contents to the left of the head, q is the current state, and v are the tape contents under and to the right of the head, including blanks for cells that have not yet been visited. The initial configuration for input w is $q_0 w$ (padded with blanks as needed).

The *computation* of $\mathcal{M}$ on an input w is a tree whose nodes are the tape configurations of $\mathcal{M}$, rooted at the initial configuration and where the children of each configuration are its successor configurations (w.r.t. δ). If α is a configuration of $\mathcal{M}$, we denote by T_α the subtree rooted at α. A subtree T_α, where q is the control state of α, is called *accepting* if either

- q is accepting,
- or q is existential and there is a successor β of α such that T_β is accepting;

– or q is universal and for any successor β of α, T_β is accepting.

The problem to check whether the computation of $\mathcal{M}$ on w is accepting is well-known to be **EXPTIME**-complete [10].

Given $\mathcal{M}$ and w, we now construct an APDS $\mathcal{P}$ and configurations c, c' such that $c \in pre^*_{\mathcal{P}}(\{c'\})$ if and only if the computation of $\mathcal{M}$ on w is accepting. This proves that the backward reachability problem is **EXPTIME**-hard, and together with Theorem 1, **EXPTIME**-complete.

The stack alphabet of $\mathcal{P}$ is $Q \cup \Sigma \cup \{\#, L, N, R\}$. A run of $\mathcal{P}$ works in two phases. In the first phase, $\mathcal{P}$ begins at a configuration $\langle Start, \# \rangle$, then nondeterministically pushes a word $\#c_0\#t_1\#c_1\#t_2\#c_2\#t_3\cdots$ onto the stack, where c_0 is the initial configuration of $\mathcal{M}$ on w, $c_1, c_2, \ldots$ are arbitrary configurations of $\mathcal{M}$, and t_i, $i \geq 1$, represents a transition of $\mathcal{M}$ from c_{i-1} to c_i. A transition t_i is constructed as follows: After pushing c_{i-1}, $\mathcal{P}$ first chooses a pair $(q, a) \in Q \times \Sigma$ and writes qa to the stack. Then, if q is accepting, $\mathcal{P}$ goes to a control state *Test* and enters the second phase (see below). If q is existential, $\mathcal{P}$ nondeterministically chooses a triple (q', a', m) from $\delta(q, a)$, pushes $q'a'm$ to the stack and continues with c_i. If q is universal, and $|\delta(q, a)| = n$, then $\mathcal{P}$ executes an alternating rule with branching degree n, where each branch writes a distinct element of $\delta(q, a)$ to the stack, and then continues with c_i.

Notice that in the first phase, there is no guarantee that c_{i-1}, t_i, and c_i are correctly related to each other. Consider a run of $\mathcal{P}$ where each branch has entered the control state *Test*, and assume (for a moment) that the choices of subsequent configurations and transitions along each branch correctly represent steps in $\mathcal{M}$. Since the branching behaviour of the run corresponds to the branching behaviour of the transitions chosen along each branch, and each branch has entered an accepting state, such a run is possible if and only if the computation of $\mathcal{M}$ on w is accepting. All that remains is to check (on each branch) whether the configurations and transitions are correctly related to each other. This is done in the second phase.

In the second phase, $\mathcal{P}$ checks (for each branch) whether the following holds:

(i) for each pair c_{i-1}, t_i, where $i \geq 1$ and $t_i = qa\ldots$, c_{i-1} has the form $uqav$ for some $u, v \in \Sigma^*$;
(ii) for each triple c_{i-1}, $t_i = qaq'a'm$, c_i, where $i \geq 1$, the control state in c_i is q', and its position is correct w.r.t. the position of the control state in c_{i-1} and m;
(iii) for each triple c_{i-1}, t_i, c_i, where $i \geq 1$, and each $j \in \{1, \ldots, p(|w|)\}$, the symbol on the j-th tape cell in c_i is correct w.r.t. the j-th tape symbol in c_{i-1} and t_i.

To perform these checks, the second phase can be seen to consist of a ‘popping thread’ that pops the stack contents and forks off a ‘checking thread’ at each position where a check is required. The popping thread can be implemented by an alternating rule of the kind $\langle Test, \gamma \rangle \hookrightarrow \{\langle Test, \varepsilon \rangle, \langle Check, \gamma \rangle\}$, where $\langle Test, \varepsilon \rangle$ is the continuation of the ‘popping thread’, and $\langle Check, \gamma \rangle$ is the beginning of the ‘checking thread’.

The checking thread for condition (i) is simple to implement: the thread pops (q, a) from the stack, enters a control state $\mathit{Check}_1(q, a)$, then removes the configuration c_{i-1} from the stack, checking whether condition (i) is met.

The checking thread for condition (ii) is similar, except for the fact that the thread needs a counter up to $p(|w|)$ to check that the position is correct.

The checking thread for condition (iii) remembers the symbol of c_i at position j and whether the head is at position $j-1$, j, $j+1$, or somewhere else, then removes the rest of c_i and reads t_i. From this information it can conclude which symbol position j in c_{i-1} should have had. It then removes part of c_{i-1} up to position j and checks whether it contains the correct symbol. Again, a counter up to $p(|w|)$ is needed.

Assume that all successful checking threads continue to remove the stack contents and then enter a control state *End*, and that the same holds for the popping thread. Then, all threads become $\langle \mathit{End}, \varepsilon\rangle$ if and only if all branches of $\mathcal{P}$ in the first phase represented correct computations of $\mathcal{M}$. Putting it differently, $\langle \mathit{Start}, \#\rangle \in \mathit{pre}^*{}_{\mathcal{P}}(\{\mathit{End}, \varepsilon\})$ if and only if the computation of $\mathcal{M}$ on w is accepting. The number of control states in $\mathcal{P}$ is $\mathcal{O}(p(|w|) \cdot |Q| \cdot |\Sigma|)$. ☐

Theorem 3. *Let $\mathcal{P} = (P, \Gamma, \Delta)$ and R be a good instance, and let $\mathcal{A}$ be a nondeterministic automaton recognizing R. Assume w.l.o.g. that $\mathcal{A}$ has one single final state. Then, the automaton resulting from the modified saturation procedure is nondeterministic and recognizes the same language as $\mathcal{A}_{pre^*}$.*

Proof. Because of the modification in the rule, the modified saturation procedure never adds an alternating rule, and so it yields a nondeterministic automaton.

We claim that if $\mathcal{A}_{pre^*}$ contains a transition $p \xrightarrow{\gamma}_{i+1} (P_1 \cup \ldots \cup P_m)$ such that $P_1 \cup \ldots \cup P_m$ is not a singleton, then at least one P_i contains a redundant state, i.e., a state from which no word can be accepted. It follows that the transition need not be added.

To prove the claim, observe that $p \xrightarrow{\gamma}_{i+1} (P_1 \cup \ldots \cup P_m)$ is obtained from some rule $\langle p, \gamma\rangle \hookrightarrow \{\langle p_1, w_1\rangle, \ldots, \langle p_n, w_m\rangle\} \in \Delta$ and a set of paths $p_1 \xrightarrow{w_1}_i P_1, \ldots, p_m \xrightarrow{w_m}_i P_m$ in $\mathcal{A}_i$. Let q_f be the final state of $\mathcal{A}$. If $P_i \neq \{q_f\}$ for some $i \in \{1, \ldots, m\}$, then P_i contains a non-final state q of $\mathcal{A}$. If q is non-redundant in $\mathcal{A}_{pre^*}$ then $\mathcal{A}_{pre^*}$ recognizes a word $p_i w_i w$ where $w \neq \varepsilon$. But then $p_i w_i w \in \mathit{pre}^*(R)$, contradicting the assumption that $\mathcal{P}, R$ is a good instance. ☐

Lemma 2. *The modified Algorithm 1 takes $O(|\delta_0| + |\Delta_0| + (|\Delta_1| + |\Delta_a|)n + |\Delta_2|n^2)$ time, where $n = |P_\varepsilon| + |Q_{ni}|$, when applied to a good instance.*

Proof. The proof is similar to the one of lemma 1. Let ν be the set of states such that for every transition $(q, \gamma, q') \in \mathit{trans}$ at any point of the algorithm, $q' \in \nu$. Line 2 adds $|P_\varepsilon| + |Q_{ni}|$ elements to ν. Since no other lines add more elements to ν, at the end of the algorithm, the size of ν is $|P_\varepsilon| + |Q_{ni}| = n$. Therefore, the resulting *rel* contains at most n possible states in its right-hand side.

At line 18, Δ' has size $O(|\Delta_2|n)$. Line 21 considers alternating rules, which can only come from line 3, so it contributes $O(|\Delta_a|)$ elements to Δ'. Line 15 is executed $O(|\Delta_2|n^2)$ times, since q', γ_2 are fixed. Also, the modification of line 9 causes line 10 to be executed $O(|\Delta_1|n + |\Delta_2|n^2 + |\Delta_a|n)$ times. Totally, additions to *trans* are no more than $O(|\delta_0| + |\Delta_0| + (|\Delta_1| + |\Delta_a|)n + |\Delta_2|n^2)$ in number, and this concludes the time complexity. $\square$