Faster Pushdown Reachability Analysis with Applications in Network Verification[⋆]

Peter Gjøl Jensen[1], Stefan Schmid[2], Morten Konggaard Schou[1], Jiří Srba[1], Juan Vanerio[2], and Ingo van Duijn[1]

[1] Department of Computer Science, Aalborg University, Aalborg, Denmark
[2] Faculty of Computer Science, University of Vienna, Vienna, Austria

Abstract. Reachability analysis of pushdown systems is a fundamental problem in model checking that comes with a wide range of applications. We study performance improvements of pushdown reachability analysis and as a case study, we consider the verification of the policy-compliance of MPLS (Multiprotocol Label Switching) networks, an application domain that has recently received much attention. Our main contribution are three techniques that allow us to speed up the state-of-the-art pushdown reachability tools by an order of magnitude. These techniques include the combination of classic *pre** and *post** saturation algorithms into a dual-search algorithm, an on-the-fly technique for detecting the possibility of early termination, as well as a counter-example guided abstraction refinement technique that improves the performance in particular for the negative instances where the early termination technique is not applicable. As a second contribution, we describe an improved translation of MPLS networks to pushdown systems and demonstrate on an extensive set of benchmarks of real internet wide-area networks the efficiency of our approach.

1 Introduction

Pushdown systems are a widely-used formalism with applications in, e.g., interprocedural control-flow analysis of recursive programs [7, 10] and model checking [3, 11, 20, 21]. Pushdown systems have recently also received attention in the context of communication networks. Modern communication networks rely on increasingly complex router configurations which are difficult to manage by human administrators. Indeed, over the last years, several major network outages were due to human errors [1, 2, 8, 15], and researchers are hence developing more automated and formal approaches to ensure policy compliance in networks. In particular, pushdown systems have been shown to enable fast automated what-if analysis of the policy compliance of an important and widely-deployed type of network, namely Multiprotocol Label Switching (MPLS) networks [18].

We are motivated by the objective to improve the performance of reachability analysis in pushdown systems, which typically relies on automata-theoretic

[⋆] Research supported by the Vienna Science and Technology Fund (WWTF), ICT19-045 (WHATIF), and the DFF project QASNET.

approach for computing the *pre** and *post** of a regular set of pushdown configurations [19]. Time is the most critical performance aspect of reachability analysis in general, and in particular, in the context of the increasingly large communication networks that need to be frequently reconfigured.

Our Contributions. We show that there is a significant potential to improve the state-of-the-art in reachability analysis of pushdown systems. In particular, we propose a fast on-the-fly early termination technique as well as an algorithm that provides a novel combination of the classic *pre** and *post** algorithms in order to harvest the benefits of both methods. We also suggest a specialization of the counter-example guided abstraction refinement (CEGAR) [5] technique that leverages equivalence classes on stack symbols as well as control states in order to improve the reachability analysis of MPLS networks that contain significant redundancy in the IP prefixes and produce a large number of MPLS labels (modeled as stack symbols). All techniques are general and apply to arbitrary pushdown systems, and are hence of interest in a wide range of applications. Finally, we also suggest a novel encoding approach of an MPLS communication network into a pushdown system that not only renders the pushdown analysis faster but also simpler compared to the recent approaches [13, 14, 18]. We report on our C++ prototype implementation and our empirical evaluation showing that the techniques can reduce the runtime by almost an order of magnitude compared to the state-of-the-art tools AalWiNes [14] and Moped [20].

Background and Related Work. We are motivated by the application of pushdown systems in order to perform automated what-if analysis of communication networks. In a nutshell, we consider a communication network interconnecting a set of routers which forward packets. The forwarding behavior of each router is defined by its pre-installed routing table which consists of a set of forwarding rules. To provide a dependable service, the network needs to fulfill a number of properties, such as reachability or loop-freedom, even under link failures.

Schmid and Srba recently showed in [18] that policy compliance of the widely-deployed MPLS networks can be verified in polynomial time, when overapproximating the possible link failures. Their approach leverages the fact that routing in MPLS networks is based on *label stacks*: packets contain stacks of labels which can be pushed and popped, and routers forward packets based on the top-of-stack label. Accordingly, these networks can be modelled as pushdown systems. In [13], the tool P-Rex was presented which implements the approach from [18]. P-Rex is implemented in Python, relies on the Moped model checker, and allows to verify complex network queries on network topologies with 20-30 routers in a matter of hours. The AalWiNes tool [14] is a follow-up work that improves the performance by an order of magnitude compared to P-Rex and replaces Moped with a tailored reachability engine written in C++.

In this paper, we show how to improve the performance by another order of magnitude compared to AalWiNes, by using three novel reachability techniques, including an early termination algorithm, a combined dual computation of *pre** and *post**, and a CEGAR approach. The CEGAR [5] technique was investigated in the context of symbolic pushdown systems before by Esparza et al. [9] who consider sequential (recursive) programs whose statements are given as binary decision diagrams (BDDs). However, the CEGAR application is not used to speed up the reachability analysis but to refine the abstractions of the programs. Moped [19] is a model checker for linear-time logic on pushdown systems and has been adapted to many use cases. For instance, jMoped [21] models java byte-code as symbolic pushdown systems allowing automated analysis and verification of invariant properties with Moped.

2 Preliminaries

A *Labelled Transition System (LTS)* is a triple $(S, \Sigma, \rightarrow)$ where S is the set of *states*, Σ is the set of *labels* and $\rightarrow \subseteq S \times \Sigma \times S$ is a *transition relation*. If $(s, a, s') \in \rightarrow$ then we write $s \xrightarrow{a} s'$. We also write $s \rightarrow s'$ if there is an $a \in \Sigma$ such that $s \xrightarrow{a} s'$ and let $\rightarrow^*$ be the reflexive and transitive closure of $\rightarrow$. The relation $\rightarrow^*$ can be annotated by the sequence of labels $w \in \Sigma^*$ as follows: $s \xrightarrow{\epsilon}{}^* s$ for any $s \in S$ where ε is the empty word, and $s \xrightarrow{aw}{}^* s'$ for $a \in \Sigma$ and $w \in \Sigma^*$ if $s \xrightarrow{a} s''$ and $s'' \xrightarrow{w}{}^* s'$ for some $s'' \in S$.

Definition 1. *A* Nondeterministic Finite Automaton (NFA) *is a tuple* $\mathcal{N} = (Q, \Sigma, \rightarrow, I, F)$ *where* Q *is a finite set of* states*,* Σ *is a finite* input alphabet*,* $\rightarrow \subseteq Q \times (\Sigma \cup \{\varepsilon\}) \times Q$ *is the* transition relation*,* $I \subseteq Q$ *is the set of* initial states*, and* $F \subseteq Q$ *is the set of* accepting states*.*

An NFA $\mathcal{N}$ *accepts* a word $w \in \Sigma^*$ if the LTS $(Q, \Sigma, \rightarrow)$ satisfies $q_0 \xrightarrow{w}{}^* q_f$ for an initial state $q_0 \in I$ and an accepting state $q_f \in F$. The language $Lang(\mathcal{N})$ is the set of all words that $\mathcal{N}$ accepts.

Definition 2. *A* Pushdown System (PDS) *is a tuple* $\mathcal{P} = (P, \Gamma, \Delta)$*, where* P *is a finite set of* control locations (states)*,* Γ *is a* stack alphabet*, and the set of* rules Δ *is a finite subset of* $(P \times \Gamma) \times (P \times \Gamma^*)$*. If* $((p, \gamma), (p', w)) \in \Delta$ *then we write* $\langle p, \gamma \rangle \hookrightarrow_{\mathcal{P}} \langle p', w \rangle$*.*

A *configuration* of a pushdown system is a pair $\langle p, w \rangle$ where $p \in P$ and $w \in \Gamma^*$. The set of all configurations is denoted $Conf(\mathcal{P})$. The semantics of a pushdown system $\mathcal{P}$ is given by the LTS $\mathcal{T}_{\mathcal{P}} = (Conf(\mathcal{P}), \Delta, \Rightarrow_{\mathcal{P}})$ where $\langle p, \gamma w' \rangle \overset{r}{\Rightarrow}_{\mathcal{P}} \langle p', ww' \rangle$ for all $w' \in \Gamma^*$ whenever there is $r = ((p, \gamma), (p', w)) \in \Delta$. If $\mathcal{P}$ is clear from the context, we may omit it from $\hookrightarrow_{\mathcal{P}}$ and $\Rightarrow_{\mathcal{P}}$. We only consider normalized PDS in which all rules $\langle p, \gamma \rangle \hookrightarrow \langle p', w \rangle$ satisfy $|w| \leq 2$. Note that any PDS can be normalized by adding at most $\mathcal{O}(|P|)$ auxiliary states [19].

Definition 3. *Let* $\mathcal{P} = (P, \Gamma, \Delta)$ *be a PDS. A* $\mathcal{P}$-automaton *is an NFA* $\mathcal{A} = (Q, \Gamma, \rightarrow, P, F)$ *with the stack symbols of* $\mathcal{P}$ *as its input alphabet and with the initial states being the control locations of* $\mathcal{P}$*.*

A $\mathcal{P}$-automaton accepts a pushdown configuration $\langle p, w \rangle$ of $\mathcal{P}$ if $p \xrightarrow{w}{}^* q$ for some $q \in F$. The set of all configurations accepted by $\mathcal{A}$ is denoted by $Lang(\mathcal{A})$. A set of configurations is called *regular* if it is accepted by some $\mathcal{P}$-automaton.

Problem 1 (Pushdown Reachability Problem). For a PDS $\mathcal{P}$ and two regular sets of configurations C and C', is there $c \in C$ and $c' \in C'$ such that $c \overset{\sigma}{\Rightarrow}{}^*_{\mathcal{P}} c'$ for some sequence of rules σ? In the affirmative case return a witness trace (c, σ).

Given a PDS $\mathcal{P}$ and a set of configurations $C \subseteq \mathit{Conf}(\mathcal{P})$ the *predecessors* are defined as $\mathit{pre}^*(C) = \{c \mid \exists c' \in C, c \Rightarrow^* c'\}$ and the *successors* as $\mathit{post}^*(C) = \{c \mid \exists c' \in C, c' \Rightarrow^* c\}$. If C is a regular set of configurations, then both $\mathit{pre}^*(C)$ and $\mathit{post}^*(C)$ are also regular sets of configurations [4].

Construction of pre^*. Given a $\mathcal{P}$-automaton $\mathcal{A} = (Q, \Gamma, \rightarrow_0, P, F)$, we construct a $\mathcal{P}$-automaton $\mathcal{A}_{\mathit{pre}^*} = (Q, \Gamma, \rightarrow, P, F)$ where $\rightarrow$ is obtained by repeatedly adding transitions to $\rightarrow_0$ according to the following saturation rule: if $\langle p, \gamma \rangle \hookrightarrow \langle p', w \rangle$ and $p' \xrightarrow{w} q$ in the current automaton, add a transition $p \xrightarrow{\gamma} q$.

Theorem 1 ([3, 12, 19]). *An automaton $\mathcal{A}_{\mathit{pre}^*}$ that satisfies $\mathit{Lang}(\mathcal{A}_{\mathit{pre}^*}) = \mathit{pre}^*(\mathit{Lang}(\mathcal{A}))$ can be built in $\mathcal{O}(|Q|^2 \cdot |\Delta|)$ time and $\mathcal{O}(|Q| \cdot |\Delta| + |\rightarrow_0|)$ space.*

There is a slightly more complicated saturation procedure for $\mathcal{A}_{\mathit{post}^*}$.

Theorem 2 ([3, 12, 19]). *An automaton $\mathcal{A}_{\mathit{post}^*}$ that satisfies $\mathit{Lang}(\mathcal{A}_{\mathit{post}^*}) = \mathit{post}^*(\mathit{Lang}(\mathcal{A}))$ can be built in $\mathcal{O}(|P| \cdot |\Delta| \cdot (n_1 + n_2) + |P| \cdot |\rightarrow_0|)$ time and space, where $n_1 = |Q \setminus P|$ and n_2 is the number of different pairs (p, γ) such that there is a rule of the form $\langle p', \gamma' \rangle \hookrightarrow \langle p, \gamma\gamma'' \rangle$ in Δ.*

Problem 1 can now be solved in polynomial time using either the *pre** or *post** algorithm by computing e.g. $\mathit{pre}^*(C')$ and checking if $C \cap \mathit{pre}^*(C') \neq \emptyset$, similarly for *post**, relying on the fact that regular languages are closed under intersection. A witness trace σ can be computed by storing metadata during the saturation procedures (see e.g. [19] for details).

3 Formal Model of MPLS Networks

An MPLS network consists of a topology and forwarding rules.

Definition 4. *A* network topology *is a directed multigraph (V, E, s, t) where V is a set of* routers*, E is a set of* links *between routers, $s : E \rightarrow V$ assigns the* source router *to each link, and $t : E \rightarrow V$ assigns the* target router*.*

We assume that links in the network can fail. This is modelled by a set $F \subseteq E$ of *failed* links. A link is *active* if it belongs to $E \setminus F$.

For a nonempty set of MPLS labels L, we define the set of *MPLS operations* on packet headers as $Op(L) = \{\mathtt{swap}(\ell) \mid \ell \in L\} \cup \{\mathtt{push}(\ell) \mid \ell \in L\} \cup \{\mathtt{pop}\}$. We define the semantics of MPLS operations $[\cdot] : Op(L) \rightarrow (L \rightarrow L^*)$ by $[\mathtt{pop}](\ell) = \varepsilon$, $[\mathtt{swap}(\ell')](\ell) = \ell'$ and $[\mathtt{push}(\ell')](\ell) = \ell'\ell$ for all $\ell, \ell' \in L$.

The forwarding of a packet in an MPLS network depends on the interface (link) that the packet arrives on, which determines the forwarding table used, and the top MPLS label in the packet header, which is used for lookup in the forwarding table. When a packet enters the MPLS domain, it does not yet have any MPLS label, and the forwarding depends only on the link that it arrives on as well as the type of the protocol that is used for the packet forwarding (this is abstracted away by the use of nondeterminism).

Definition 5. *An* MPLS network *is a tuple $N = (V, E, s, t, L, \tau)$ where (V, E, s, t) is a network topology, L is a finite set of MPLS labels, and $\tau : E \cup (E \times L) \to \left(2^{E \times Op(L)^+}\right)^*$ is the routing table.*

For every link $e \in E$ and for every link-label pair $(e, \ell) \in E \times L$, the routing table returns a sequence of *traffic engineering groups* $O_1 O_2 \dots O_n$ where each group is a set of the form $\{(e_1, \omega_1), \dots, (e_m, \omega_m)\}$ where e_j is the outgoing link such that $t(e) = s(e_j)$ and $\omega_j \in Op(L)^+$ is a nonempty sequence of MPLS operations to be performed on the packet header. Figure 1a gives an example of an MPLS network with its routing table in Figure 1b. Here the priority column refers to the index of the corresponding traffic engineering group.

The semantics of a traffic engineering group is that any pair of active link and operation sequence in the group can be nondeterministically chosen, hence abstracting away from various specific routing policies that allow e.g. splitting a flow along multiple paths. The group O_i has a higher priority than O_{i+1}, and during forwarding the router always selects the traffic engineering group with the highest priority and at least one active link.

For a traffic engineering group $O = \{(e_1, \omega_1), (e_2, \omega_2), \dots, (e_m, \omega_m)\}$ let $E(O) = \{e_1, e_2, \dots, e_m\}$ denote the set of outgoing links in the group.

Definition 6. *For a set of failed links $F \subseteq E$ we define the* active routing table *$\tau_F : E \cup (E \times L) \to 2^{E \times Op(L)^+}$ as $\tau_F(u) = \{(e', \omega) \in \mathcal{A}_F(\tau(u)) \mid e' \in E \setminus F\}$, where $u = e$ or $u = (e, \ell)$ and $\mathcal{A}_F$ is the* active traffic engineering group *defined as $\mathcal{A}_F(O_1 O_2 \dots O_n) = O_j$ if j is the lowest index such that $E(O_j) \setminus F \neq \emptyset$, or $\mathcal{A}(O_1 O_2 \dots O_n) = \emptyset$ if no such j exists.*

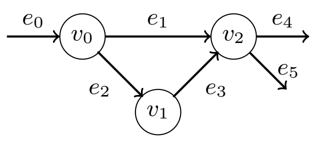
Definition 7. *The semantics of MPLS operations is a partial* header rewrite function *$\mathcal{H} : L^* \times Op(L)^* \rightharpoonup L^*$, where $\omega, \omega' \in Op(L)^*$, $h \in L^*$ and ε is the empty sequence of operations:*

$$\mathcal{H}(h, \omega) = \begin{cases} h & \text{if } \omega = \varepsilon \\ \mathcal{H}([op](\ell) \circ h', \omega') & \text{if } \omega = op \circ \omega' \text{ and } h = \ell \circ h' \text{ with } \ell \in L, h' \in L^* \\ \text{undefined} & \text{otherwise .} \end{cases}$$

Using the example from Figure 1, the operation sequence $\texttt{swap}(12) \circ \texttt{push}(20)$ applied to the header $10 \circ 30$ yields $\mathcal{H}(10 \circ 30, \texttt{swap}(12) \circ \texttt{push}(20)) = 20 \circ 12 \circ 30$.

Definition 8. *A* trace *in a network $N = (V, E, s, t, L, \tau)$, given a set of failed links $F \subseteq E$, is any finite sequence $(e_1, h_1)(e_2, h_2) \dots (e_n, h_n) \in \left((E \setminus F) \times L^*\right)^*$ of link-header pairs where for all i, $1 \leq i < n$, $h_{i+1} = \mathcal{H}(h_i, \omega)$ for some $(e_{i+1}, \omega) \in \tau_F(u)$, where either $u = e_i$ or $u = (e_i, head(h_i))$, where $head(h_i)$ is the top (left-most) label of h_i. If $h_i = \varepsilon$ then $head(h_i)$ is undefined.*

In Figure 1c we can see a trace σ_1 in the network without any failed links, while for the failure set $F = \{e_1\}$ we notice that σ_1 is not a trace, while σ_2 is.



(a) Network topology

Router	e_{in}	**Label**	**Priority**	e_{out}	**Operation**
v_0	e_0	–	1	e_1	push(11)
	e_0	–	2	e_2	push(11) ∘ push(20)
	e_0	10	1	e_1	swap(12)
	e_0	10	2	e_2	swap(12) ∘ push(20)
v_1	e_2	20	1	e_3	pop
v_2	e_1	11	1	e_4	pop
	e_1	12	1	e_5	pop
	e_3	11	1	e_4	pop
	e_3	12	1	e_5	pop

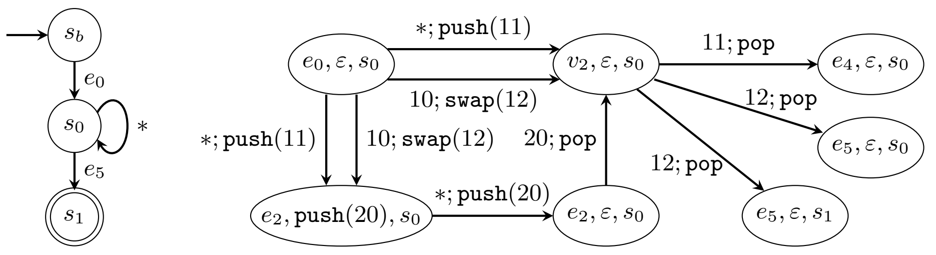
(b) Routing table

$\sigma_1 = (e_0, \varepsilon)(e_1, 11)(e_4, \varepsilon)$ for $F = \emptyset$

$\sigma_2 = (e_0, 10 \circ 30)(e_2, 20 \circ 12 \circ 30)(e_3, 12 \circ 30)(e_5, 30)$ for $F = \{e_1\}$

$\varphi = \langle 10 \cdot^* \rangle \; e_0 \; \cdot^* \; e_5 \; \langle 30 \rangle \; 1$ is satisfied by σ_2

(c) Example traces σ_1 and σ_2 under a set of failed links F, and an example query φ



(d) Corresponding pushdown system for the query φ. Left: the NFA $\mathcal{N}_b$ for φ. Right: the generated PDS $\mathcal{P}$. The labelled arrow $(p) \xrightarrow{\ell;op} (p')$ denotes the rule $\langle p, \ell \rangle \hookrightarrow \langle p', [op](\ell) \rangle$. The state (v_2, ε, s_0) is merged from (e_1, ε, s_0) and (e_3, ε, s_0).

$P_i = \{(e_0, \varepsilon, s_0)\}$

$P_f = \{(e_5, \varepsilon, s_1)\}$

$Lang(\mathcal{N}_i) = \{10 \circ w \circ \bot \mid w \in L^*\}$

$Lang(\mathcal{N}_f) = \{30 \circ \bot\}$

(e) Initial/final configurations for φ

$\langle (e_0, \varepsilon, s_0), 10 \circ 30 \circ \bot \rangle \Rightarrow_{\mathcal{P}}$

$\langle (e_2, \texttt{push}(20), s_0), 12 \circ 30 \circ \bot \rangle \Rightarrow_{\mathcal{P}}$

$\langle (e_2, \varepsilon, s_0), 20 \circ 12 \circ 30 \circ \bot \rangle \Rightarrow_{\mathcal{P}}$

$\langle (v_2, \varepsilon, s_0), 12 \circ 30 \circ \bot \rangle \Rightarrow_{\mathcal{P}} \langle (e_5, \varepsilon, s_1), 30 \circ \bot \rangle$

(f) Computation in PDS $\mathcal{P}$ corresponding to σ_2

Fig. 1: Example of a small network and its encoding into a pushdown system

3.1 MPLS Network Verification

Similar to prior work [13, 14], we present a powerful query language that allows us to specify regular trace properties, both regarding the initial and final label-stacks as well as the sequence of links in the trace.

Definition 9. *A reachability* query *for an MPLS network $N = (V, E, s, t, L, \tau)$ is of the form $\langle a \rangle \; b \; \langle c \rangle \; k$ where a and c are regular expressions over the set of labels L, b is a regular expression over the set of links E, and $k \geq 0$ specifies the maximum number of failures to be considered.*

We assume here a standard syntax for regular expressions and by $Lang(a)$, $Lang(b)$ and $Lang(c)$ we understand the regular language defined by the expressions a, b and c, respectively. Intuitively, the query $\langle a \rangle\ b\ \langle c \rangle\ k$ asks if there is a network trace such that the initial header (stack of labels) belongs to $Lang(a)$, the sequence of visited links belongs to $Lang(b)$ and at the end of the trace the final header belongs to $Lang(c)$.

We further use the following notation for specifying links in the network. If v and u are routers, then $[v\#u]$ matches any link e from v to u such that $s(e) = v$ and $t(e) = u$. The dot-syntax is used to denote any link or label in the network and it is extended to match also any router so that $[v\#\cdot] = \bigcup_{u \in V} [v\#u]$ and $[\cdot\#u] = \bigcup_{v \in V} [v\#u]$.

Problem 2 (Query Satisfiability Problem). Given an MPLS network N and a query $\varphi = \langle a \rangle\ b\ \langle c \rangle\ k$, decide if there exists a trace $\sigma = (e_1, h_1) \dots (e_n, h_n)$ in the network N for some set of failed links F such that $|F| \leq k$ where $h_1 \in Lang(a)$, $e_1 \dots e_n \in Lang(b)$, and $h_n \in Lang(c)$. If this is the case, the query φ is *satisfied* and we call σ a *witness trace.*

In Figure 1c the query φ asks if a packet with the top most label 10 can be forwarded from the link e_0 to e_5, while just leaving the label 30 on the label-stack. This query is satisfied and the trace σ_2 serves as a witness trace. On the other hand, the query $\langle\ \cdot^* \rangle\ [\cdot\#v_1]\ \cdot^*\ e_3\ \langle\ \cdot^* \rangle\ 0$ is not satisfied as it asks if a packet (with any header) arriving on some link to the router v_1 (note that e_0 is the only such link) can reach the link e_3 if no links fail. Such a trace exists only if we allow for at least one failed link.

3.2 From Query Satisfiability to Pushdown Reachability

We now solve the query satisfiability problem by translation to the pushdown reachability problem. This is an over-approximation, so in a few cases a positive result cannot be transfered back to the query satisfiability problem. Notice that our construction is different from the one in [13]. In particular, we model the initial and final headers directly as NFA rather than simulating them with PDSs, which makes the reduction simpler and more efficient at the same time.

The behavior of the network for a fixed set of failed links F is given by the active routing table τ_F, however to represent the possible behavior for any set of failed links F with $|F| \leq k$, we use the following definition.

Definition 10. *For a network $N = (V, E, s, t, L, \tau)$ and number k, we define the* overapproximating routing table $\tau^k(u) = \bigcup_{j=1}^{i} O_j$*, where $\tau(u) = O_1 O_2 \dots O_n$ and i is the smallest index such that $|\bigcup_{j=1}^{i} E(O_j)| > k$.*

The routing table τ^k overapproximates all possible routing table entries if up to k links fail at any router.

Given a network $N = (V, E, s, t, L, \tau)$ and a query $\varphi = \langle a \rangle\ b\ \langle c \rangle\ k$, let $\mathcal{N}_a = (S_a, L, \rightarrow_a, \{s_a\}, F_a)$, $\mathcal{N}_b = (S_b, E, \rightarrow_b, \{s_b\}, F_b)$ and $\mathcal{N}_c = (S_c, L, \rightarrow_c, \{s_c\}, F_c)$ be the NFAs corresponding to the regular expressions a, b and c. Let $L_\perp =$

$L \cup \{\bot\}$ where $\bot$ is used to represent the bottom of the stack. We construct a PDS $\mathcal{P} = (P, L_\bot, \Delta)$ where $P = E \times \overline{Ops} \times S_b$ and $\overline{Ops}$ is the set of all operation sequences and suffixes hereof occurring in τ^k. The set of rules Δ is defined by:

a) $\langle (e, \varepsilon, s), \ell\rangle \hookrightarrow \langle (e', \omega, s'), [op](\ell)\rangle$ if $s \xrightarrow{e'}{}_b^* s'$ and $(e', op \circ \omega) \in \tau^k(u)$ where either (i) $u = (e, \ell)$, or (ii) $u = e$, $\ell \in L$, or (iii) $u = e$, $\ell = \bot$, $op = \mathtt{push}(\ell')$.
b) $\langle (e, op \circ \omega, s), \ell\rangle \hookrightarrow \langle (e, \omega, s), [op](\ell)\rangle$ for $\ell \in L$ and for $\ell = \bot$ if $op = \mathtt{push}(\ell')$.

Finally, we define the initial states $P_i = \{(e, \varepsilon, s) \mid e \in E, s \in S_b, s_b \xrightarrow{e}{}_b^* s\}$, and the final states $P_f = \{(e, \varepsilon, s_f) \mid e \in E,\ s_f \in F_b\}$. Let $\mathcal{N}_\bot$ be an NFA such that $\mathit{Lang}(\mathcal{N}_\bot) = \{\bot\}$. Let $\mathcal{N}_i = \mathcal{N}_a \circ \mathcal{N}_\bot$ and $\mathcal{N}_f = \mathcal{N}_c \circ \mathcal{N}_\bot$ where $\circ$ is the standard NFA concatenation operator. For the running example this is shown in Figure 1e. Now the query satisfiability problem is reduced to the problem of finding configurations $c \in P_i \times \mathit{Lang}(\mathcal{N}_i)$ and $c' \in P_f \times \mathit{Lang}(\mathcal{N}_f)$ such that $c \overset{\sigma}{\Rightarrow}{}_{\mathcal{P}}^* c'$, and in the positive case outputing the trace (c, σ).

Optimizations. To reduce the size of the PDS we use the following optimizations. We merge control locations (e, ω, s) and (e', ω, s) for which $t(e) = t(e')$, $\tau(e) = \tau(e')$ and $\tau(e, \ell) = \tau(e', \ell)$ for all $\ell \in L$, i.e. the lookup is independent of which interface on the router the packet arrives on, which is often the case in many existing networks. We only construct control states that are reachable from P_i. If a rule $\langle p, \ell\rangle \hookrightarrow \langle p', [op](\ell)\rangle$ is added for all $\ell \in L_\bot$, we represent it succinctly as $\langle p, *\rangle \hookrightarrow \langle p', [op](*)\rangle$ where $*$ is a wildcard representing any label. The wildcard can be handled efficiently by our post^* algorithm, while for pre^* it needs to be unfolded. In Figure 1d we can see the generated pushdown system for our running example and in Figure 1f we show an execution of the pushdown system corresponding to the network trace σ_2.

We can now show that if there is a network trace satisfying a given query then the constructed pushdown system provides a positive answer in the reachability analysis.

Theorem 3. *Given a network N and a query φ, if there exists a witness trace in the network satisfying φ, then there exist $c \in P_i \times \mathit{Lang}(\mathcal{N}_i)$, $c' \in P_f \times \mathit{Lang}(\mathcal{N}_f)$ and $\sigma \in \Delta^*$ such that $c \overset{\sigma}{\Rightarrow}{}_{\mathcal{P}}^* c'$.*

Proof (Sketch). By induction on the length of the witness trace we construct the corresponding pushdown execution following the construction of the pushdown rules Δ. One step in the network trace can be simulated by a sequence of pushdown transitions as the rules of type b) apply the MPLS operations sequentially one by one. □

For the other direction, we have to first make sure that the trace obtained from the execution in the pushdown system is indeed a valid network trace (since the pushdown system overapproximates the set of all valid traces as it assumes that at any router, up to k links can fail).

Reconstruction of Network Traces. The reachability analysis for the pushdown system $\mathcal{P}$ returns (in the affirmative case) a trace $\langle p_0, w_0\rangle \overset{r_1}{\Rightarrow}_{\mathcal{P}} \ldots \overset{r_m}{\Rightarrow}_{\mathcal{P}} \langle p_m, w_m\rangle$. We extract (e, h) for every i such that $p_i = (e, \omega, s)$ and $w_i = h \circ \bot$ where $\omega = \varepsilon$, producing a network trace $(e_0, h_0) \ldots (e_n, h_n)$. For each rule r of type a) that was added due to $(e', \omega) \in \tau^k(u)$, we define $F^\tau(r) = \bigcup_{j=1}^{i-1} E(O_j)$ where $\tau(u) = O_1 O_2 \ldots$ and i is the smallest index such that $(e', \omega) \in O_i$. Let $F = \bigcup_{i=1}^{n} F^\tau(r_i)$ be the set of failed links in order to enable the execution of the trace. Now we have to check that $\{e_0, \ldots, e_n\} \cap F = \emptyset$ and $|F| \leq k$ in order to guarantee that the corresponding network trace is executable; otherwise the overapproximation returns an inconclusive answer.

Theorem 4. *Given a network N and a query φ, if in the constructed pushdown system there exist $c \in P_i \times Lang(\mathcal{N}_i)$, $c' \in P_f \times Lang(\mathcal{N}_f)$ and $\sigma \in \Delta^*$ s.t. $c \overset{\sigma}{\Rightarrow}{}^*_{\mathcal{P}} c'$ from which a valid network trace σ' can be reconstructed, then σ' satisfies φ.*

Proof (Sketch). From the construction of the pushdown system and the encoding of MPLS operations by a series of pushdown transitions, we can see that if the reconstructed trace only uses active links, i.e. $\{e_0, \ldots, e_n\} \cap F = \emptyset$, then it corresponds to a correct network trace for the routing table τ^k. However, as τ^k allows for up to k link failures at any router along the trace, the total number of failed links along the reconstructed trace may exceed the bound k. This is detected in the trace reconstruction procedure. □

4 Improving Pushdown System Reachability Analysis

We now describe our improvements to the pushdown reachability analysis.

4.1 Early Termination of Reachability Algorithms

In Section 2 we showed that for a given PDS $\mathcal{P} = (P, \Gamma, \Delta)$ and $\mathcal{P}$-automaton $\mathcal{A}$ that represents a set of configurations in $\mathcal{P}$, we can construct the $\mathcal{A}_{post^*}$ and $\mathcal{A}_{pre^*}$ automata by iteratively adding additional transitions to the existing automaton $\mathcal{A}$. During this saturation procedure, the language of the current $\mathcal{P}$-automaton $\mathcal{A}$ can only increase (w.r.t. subset inclusion). Hence if at any point the current $\mathcal{P}$-automaton has a nonempty intersection with some set of target configurations, it will have the nonempty intersection also after the saturation procedure terminates. We can hence allow for an early termination as we can return a witness trace before completing the saturation procedure.

We further generalize this idea by considering $\mathcal{P}$-automata $\mathcal{A}_1 = (Q_1, \Gamma, \rightarrow_1, P, F_1)$ and $\mathcal{A}_2 = (Q_2, \Gamma, \rightarrow_2, P, F_2)$ that can be step-by-step saturated by calling (in arbitrary order) the functions AddTransition($q_1 \xrightarrow{\gamma}_1 q_1'$) and AddTransition($q_2 \xrightarrow{\gamma}_2 q_2'$), respectively. Each such call will add the corresponding transition in its argument to the automaton $\mathcal{A}_1$ resp. $\mathcal{A}_2$ and at the same time compute the reachable part (stored in the nondecreasing set R of pairs of states in $\mathcal{A}_1$ and $\mathcal{A}_2$) of the product automaton $\mathcal{A}_\cap$ representing the

Algorithm 1 On-the-fly computation of product automaton

```
    Input: $\mathcal{P}$-automata $\mathcal{A}_1 = (Q_1, \Gamma, \rightarrow_1, P, F_1)$ and $\mathcal{A}_2 = (Q_2, \Gamma, \rightarrow_2, P, F_2)$
 1: Initialize $R \subseteq Q_1 \times Q_2$ to $\emptyset$
 2: Let $\mathcal{A}_\cap \leftarrow (Q_1 \times Q_2, \Gamma, \rightarrow, \{(p,p) \mid p \in P\}, F_1 \times F_2)$ where $\rightarrow$ initially does not
    contain any transitions

 3: function AddState($q_1$, $q_2$)
 4:     if $(q_1, q_2) \notin R$ then
 5:         $R \leftarrow R \cup (q_1, q_2)$
 6:         if $q_1 \in F_1$ and $q_2 \in F_2$ then exit and return true
 7:         for all $q'_1 \in Q_1, q'_2 \in Q_2, \gamma \in \Gamma$ s.t. $q_1 \xrightarrow{\gamma}_1 q'_1$ and $q_2 \xrightarrow{\gamma}_2 q'_2$ do
 8:             add $(q_1, q_2) \xrightarrow{\gamma} (q'_1, q'_2)$ to $\mathcal{A}_\cap$
 9:             AddState($q'_1$, $q'_2$)

10: function AddTransition($q_i \xrightarrow{\gamma}_i q'_i$)                      ▷ with $i \in \{1,2\}$
11:     add $q_i \xrightarrow{\gamma}_i q'_i$ to $\mathcal{A}_i$
12:     for all $q_{3-i}, q'_{3-i} \in Q_{3-i}$ s.t. $(q_1, q_2) \in R$ and $q_{3-i} \xrightarrow{\gamma}_{3-i} q'_{3-i}$ do
13:         add $(q_1, q_2) \xrightarrow{\gamma} (q'_1, q'_2)$ to $\mathcal{A}_\cap$
14:         AddState($q'_1$, $q'_2$)
```

current intersection of $\mathcal{A}_1$ and $\mathcal{A}_2$. The function call AddTransition($q_i \xrightarrow{\gamma}_i q'_i$) where $i \in \{1, 2\}$ relies on the function AddState(q_1, q_2) given in Algorithm 1 and before any calls to AddTransition are made, it is assumed that the product automaton is initialized by calling AddState(p, p) for all states $p \in P$. The algorithm exits (early terminates) and returns true as soon as the product automaton accepts at least one string.

Proposition 1. *Let $\mathcal{A}_1$ and $\mathcal{A}_2$ be two initial $\mathcal{P}$-automata and let $\mathcal{A}'_1$ and $\mathcal{A}'_2$ be the resulting $\mathcal{P}$-automata after an arbitrary number of calls to the function* AddTransition *given in Algorithm 1. Then $Lang(\mathcal{A}_\cap) = Lang(\mathcal{A}'_1) \cap Lang(\mathcal{A}'_2)$ and as soon as $Lang(\mathcal{A}_\cap) \neq \emptyset$, the algorithm returns true.*

This on-the-fly detection of nonemptiness of the intersection between two $\mathcal{P}$-automata can be used to allow for early termination when deciding the reachability in pushdown systems using the *pre** and *post** approach described in Section 2. Here only one of the two $\mathcal{P}$-automata is saturated while the other automaton remains unchanged. We now show that this on-the-fly detection of nonemptiness can be applied, with significant performance improvements, also when both approaches are combined.

4.2 Combining Forward and Backward Search

Our experiments show that none of the two approaches, *pre** and *post**, is superior to the other one. Our aim is to further improve the reachability analysis of pushdown systems by combining these two methods into *dual*$*$ algorithm. We first observe the following facts.

Algorithm 2 Dual search

```
    Input: $\mathcal{P}$-automata $\mathcal{A}$ and $\mathcal{A}'$
 1: for $p$ in $P$ do AddState($p$, $p$)
 2: Initialize $pre^*$ algorithm for $\mathcal{A}'$ and $post^*$ for $\mathcal{A}$ (incl. worksets of transitions)
 3: while $workset_{pre^*} \neq \emptyset$ and $workset_{post^*} \neq \emptyset$ do
 4:     pop $t$ from $workset_{pre^*}$
 5:     execute one step of $pre^*$ using $t$
 6:     for $t'$ newly added to $workset_{pre^*}$ do AddTransition($t'$)     (can return true)
 7:     pop $t$ from $workset_{post^*}$
 8:     execute one step of $post^*$ using $t$
 9:     for $t'$ newly added to $workset_{post^*}$ do AddTransition($t'$)     (can return true)
10: return false
```

Proposition 2. *Given a PDS $\mathcal{P} = (P, \Gamma, \Delta)$ and regular sets C and C' of its configurations, the following statements are equivalent: a) $c \Rightarrow^* c'$ for some $c \in C$ and $c' \in C'$, b) $C \cap pre^*(C') \neq \emptyset$, c) $post^*(C) \cap C' \neq \emptyset$, and d) $post^*(C) \cap pre^*(C') \neq \emptyset$.*

Let the $\mathcal{P}$-automata $\mathcal{A}$ and $\mathcal{A}'$ represent the sets of configurations C and C', respectively. The classical approach to the reachability problem, formulated in Proposition 2a, either uses the equivalent formulation in b) and iteratively constructs $\mathcal{A}'_{pre^*}$ while checking whether its language has a nonempty intersection with the set C, or it uses part c) and constructs $\mathcal{A}_{post^*}$ while checking for nonempty intersection with C'.

We suggest a novel combination of these two methods while relying on Proposition 2d. In Algorithm 2, we (sequentially) interleave the executions of the *post** saturation procedure on $\mathcal{A}$ and the *pre** procedure on $\mathcal{A}'$. The intersection of the two automata is computed on-the-fly using Algorithm 1 where each of the saturation procedures calls its respective AddTransition function and Algorithm 2 terminates with **true** as soon as the intersection becomes nonempty. Once one of the saturation algorithms completes its execution, the algorithm returns **false**. Notice that this approach is different from running *pre** and *post** independently in parallel since our algorithm allows the two search directions to 'meet in the middle'. In Section 5 we document a gain of almost an order of magnitude compared to saturating exclusively $\mathcal{A}$ or $\mathcal{A}'$.

4.3 Abstraction Refinement for Pushdown System Reachability

We now explore an abstraction technique [6] in order to reduce the size of the verified PDS. We suggest (in a heuristic way) an initial abstraction by collapsing selected stack symbols and control states and use counter-example guided abstraction refinement [5] in case we obtain spurious traces.

Abstraction of Pushdown Model of MPLS Network. As described in Section 3.2, we consider a network $N = (V, E, s, t, L, \tau)$, NFAs that originate from the given query $\mathcal{N}_a = (S_a, L, \rightarrow_a, s_a, F_a)$, $\mathcal{N}_b = (S_b, E, \rightarrow_b, s_b, F_b)$ and $\mathcal{N}_c = (S_c, L, \rightarrow_c, s_c, F_c)$, and the overapproximating routing table τ^k.

Let $\mathbb{L}$ and $\mathbb{E}$ be the sets of abstract labels resp. edges that are possibly smaller than the sets L and E. A *network abstraction* is a surjective function $\alpha : L \cup E \rightarrow \mathbb{L} \cup \mathbb{E}$ such that $\alpha(\ell) \in \mathbb{L}$ for all $\ell \in L$ and $\alpha(e) \in \mathbb{E}$ for all $e \in E$.

Example 1. Let $\mathbb{L} = \{\bullet\}$ and $\mathbb{E} = \{\star\}$ such that $\alpha(\ell) = \bullet$ for $\ell \in L$ and $\alpha(e) = \star$ for $e \in E$. This is the coarsest abstraction that does not distinguish between any labels nor edges. On the other hand, if $\mathbb{L} = L$ and $\mathbb{E} = E$ then the abstraction $\alpha(x) = x$ for $x \in L \cup E$ is the most fine-grained one.

We extend α in a straightforward way to apply to headers and sequences of MPLS operations. We now construct an α-abstracted PDS $\mathcal{P} = (P, \mathbb{L}_\perp, \Delta)$ similar to Section 3.2 such that $P = \mathbb{E} \times \overline{\mathbb{O}ps} \times S_b$ where $\overline{\mathbb{O}ps} = \{\alpha(\omega) \mid \omega \in \overline{Ops}\}$ and Δ is defined as above except that rule of type a) now uses the abstraction:

a) $\langle(\alpha(e), \varepsilon, s), \alpha(\ell)\rangle \hookrightarrow \langle(\alpha(e'), \alpha(\omega), s'), [\alpha(op)](\alpha(\ell))\rangle$ if $(e', op \circ \omega) \in \tau^k(u)$ and $s \xrightarrow{e'}_b^* s'$ where either (i) $u = (e, \ell)$, or (ii) $u = e$ and $\ell \in L$, or (iii) $op = \mathtt{push}(\ell')$, $u = e$ and $\ell = \perp$.

We also define α-abstracted initial states $P_i = \{(\alpha(e), \varepsilon, s) \mid e \in E, s \in S_b, s_b \xrightarrow{e}_b^* s\}$ and final states $P_f = \{(\alpha(e), \varepsilon, s_f) \mid e \in E,\ s_f \in F_b\}$. Finally, we define an abstraction of an NFA $\mathcal{N} = (S, L, \rightarrow, \{s_0\}, F)$ as $\alpha(\mathcal{N}) = (S, \mathbb{L}, \rightarrow_\alpha, \{s_0\}, F)$ where $s \xrightarrow{\alpha(\ell)}_\alpha s'$ in $\alpha(\mathcal{N})$ iff $s \xrightarrow{\ell} s'$ in $\mathcal{N}$. Using this, let $\mathcal{N}_i = \alpha(\mathcal{N}_a) \circ \mathcal{N}_\perp$ and $\mathcal{N}_f = \alpha(\mathcal{N}_c) \circ \mathcal{N}_\perp$. Theorem 3 can now be shown to hold also for this α-abstracted PDS.

We now show how to reconstruct a concrete network trace from the α-abstracted pushdown trace. The reconstruction may finish with a success (a concrete network trace is found) or it suggests a refinement of the abstraction function α and the whole verification process is repeated (CEGAR).

Reconstruction of Network Traces. Given a trace $\langle p_0, w_0\rangle \overset{r_1}{\Rightarrow}_{\mathcal{P}} \ldots \overset{r_m}{\Rightarrow}_{\mathcal{P}} \langle p_m, w_m\rangle$ in the α-abstracted PDS, we take the subsequence of rules in the trace of type a), and for each such rule r_i define T_i as the set of forwarding rules (u, e', ω) such that r_i was added due to $(e', \omega) \in \tau^k(u)$.

For each set T_i, define $[T_i]$ as a mapping between sets of link-header pairs: $[T_i](C) = \bigcup_{(e,h)\in C}\{(e', h') \mid (u, e', \omega) \in T_i,\ \mathcal{H}(h, \omega) = h', \text{and } u = e \text{ or } u = (e, head(h))\}$. If $C' = [T_i](C)$ then we write $C \underset{T_i}{\Longrightarrow} C'$. The initial set of link-header pairs is $C_0 = \{(e, h) \in E \times L^* \mid p_0 = (\alpha(e), \varepsilon, s),\ s_b \xrightarrow{e}_b^* s,\ w_0 = \alpha(h) \circ \perp,\ h \in Lang(\mathcal{N}_a)\}$. The set of reachable link-header pairs is now found by $C_0 \underset{T_1}{\Longrightarrow} C_1 \underset{T_2}{\Longrightarrow} \ldots \underset{T_n}{\Longrightarrow} C_n$. If $C_n \neq \emptyset$ and there exists $(e, h) \in C_n$ such that $h \in Lang(\mathcal{N}_c)$, then we have a concrete network trace, where we finally compute and check the set of failed links against the trace as in Section 3.2. Otherwise the PDS trace is a spurious counter-example that will guide the refinement of the abstraction α.

Refinement from pushdown system rules. If $C_n = \emptyset$ then we compute the refinement based on the rules of the pushdown system: let i be such that $C_i \neq \emptyset$ and $C_{i+1} = \emptyset$, and we must have some $(e, h) \in C_i$ and $(u, e'', \omega) \in T_{i+1}$ such that $u = (e', \ell')$ and $head(h) = \ell$ where $(\alpha(e), \alpha(\ell)) = (\alpha(e'), \alpha(\ell'))$ but $(e, \ell) \neq (e', \ell')$, or that $u = e'$ where $\alpha(e) = \alpha(e')$ but $e \neq e'$. In the refined abstraction α' we ensure that for all such $(e, \ell) \neq (e', \ell')$ we have $(\alpha'(e), \alpha'(\ell)) \neq (\alpha'(e'), \alpha'(\ell'))$, and similarly for such $e \neq e'$ we have $\alpha'(e) \neq \alpha'(e')$. The refined abstraction α' should preferably be as coarse as possible. In the appendix, we present a greedy algorithm (used in our experiments) for computing one such suitable refinement.

Refinement from final headers. If $C_n \neq \emptyset$ but for all $(e, h) \in C_n$ we have $h \notin Lang(\mathcal{N}_c)$ then we compute the refinement based on the transitions in the NFA encoding the final headers: for all pairs $(e, h) \in C_n$ we must have $\alpha(h) \in Lang(\alpha(\mathcal{N}_c))$ but $h \notin Lang(\mathcal{N}_c)$. That is we have in $\alpha(\mathcal{N}_c)$: $s_c \xrightarrow{\alpha(\ell_1)}_\alpha s_1 \xrightarrow{\alpha(\ell_2)}_\alpha \dots \xrightarrow{\alpha(\ell_n)}_\alpha s_n$ with $h = \ell_1 \ell_2 \dots \ell_n$, but in $\mathcal{N}_c$: $s_c \xrightarrow{\ell_1} s_1 \xrightarrow{\ell_2} \dots \xrightarrow{\ell_i} s_i \not\xrightarrow{\ell_{i+1}}$, for some i with $i < n$. Now there must be another label ℓ' such that $\alpha(\ell_{i+1}) = \alpha(\ell')$ and $s_i \xrightarrow{\ell'} s_{i+1}$, but $\ell_{i+1} \neq \ell'$. In the refined abstraction α' we ensure that for all such ℓ' we have $\alpha'(\ell_{i+1}) \neq \alpha'(\ell')$ and we do this for all relevant h.

Heuristics for initial abstraction. We use a heuristic to construct the initial abstraction. We group labels based on their next-hop links, i.e. $\mathbb{L} \subseteq 2^E$ and $\alpha(\ell) = \{e' \mid (e', \omega) \in \tau^k(e, \ell) \text{ for some } e\}$. We group links based on their explicit mention in the path expression of the query, i.e. $\mathbb{E} \subseteq 2^{S_b \times S_b}$ and $\alpha(e) = \{(s, s') \mid s \xrightarrow{e}_b s'\}$.

5 Implementation and Experiments

We implemented the translation of MPLS networks to pushdown automata as well as the three improvements to the reachability analysis in our prototype tool written in C++. In our experimental evaluation, we use real-world network topologies from the Internet Topology Zoo [16]. We implemented a Python tool that for a given network topology distributes the MPLS labels and configures the forwarding tables by following the commonly used Label Distribution Protocol (LDP), the Resource Reservation Protocol with Traffic Engineering extensions (RSVP-TE), as well as the industry-standard MPLS VPN services. We generate the forwarding tables using four different parameter settings for the ten largest topologies from [16] (ranging from 100 nodes up to 700 nodes). This results in 40 MPLS data planes, each with 1,520 queries that are randomly instantiated from a set of query templates describing reachability, waypointing, loop-freedom, service-chaining and transparency [13], with the maximum number of failures $k \in \{0, 1, 2, 3\}$. We balance the benchmark in order to obtain an even distribution between positive and negative queries. The whole benchmark consists of 60,800 queries that are verified by each of the solvers, in particular our algorithms referred to as *post**, *pre** and *dual** (all without CEGAR), compared to the state-of-the-art pushdown reachability algorithms implemented in Moped [20] (Moped-

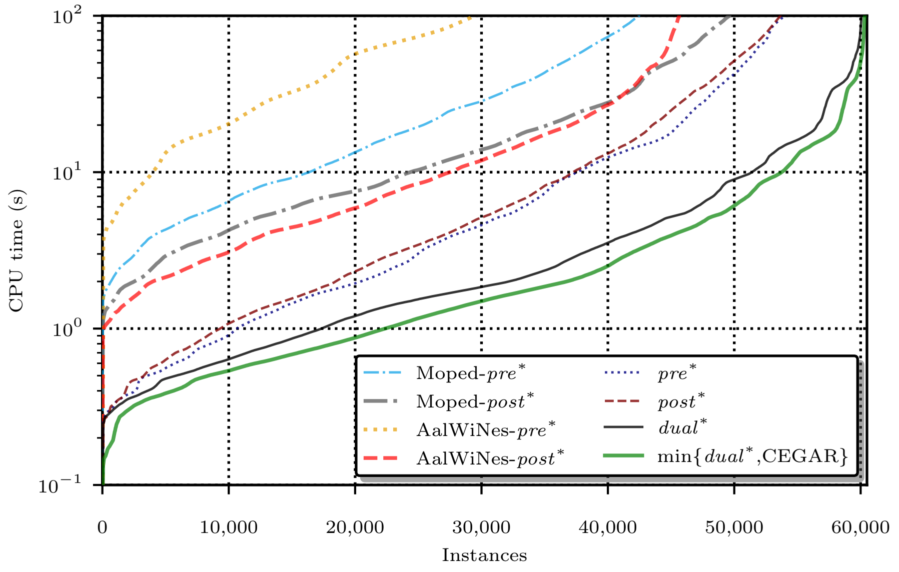


Fig. 2: Comparison of solvers; all 60,800 instances (x-axis) are for each solver independently sorted by the verification time (y-axis, note the logarithmic scale).

*pre** and Moped-*post**) and in AalWiNes [14] (AalWiNes-*pre** and AalWiNes-*post**). The experiments were run on a cluster with AMD EPYC 7551 processors at 2.55 GHz (boost disabled) with 32GB memory limit and 100 second timeout. Time spent on parsing files is excluded. The source code, experimential benchmark and all data are available at https://doi.org/10.5281/zenodo.5005893.

The results are presented in Figure 2 in terms of performance plots where all instances for the competing approaches are independently sorted by their running times and plotted on the x-axis while the y-axis contains (on logarithmic scale) the respective running times in seconds.

The performance curve for AalWiNes-*pre** and Moped-*pre** are significantly slower than the other methods, including Moped-*post** and AalWines-*post**, which are comparable. Our new improved *pre** and *post** methods are comparable performance-wise and already more than two times faster (on the median instance) compared to AalWiNes-*post**. This is mainly due to our early termination improvement and a more efficient encoding of the network.

The introduction of our *dual** approach significantly improves the performance of both *pre** and *post**, and on the median instance the *dual** solver is more than 6 times faster than the previous state-of-the-art AalWiNes-*post** approach, while the curves further open with the increasing complexity of the reachability problems. On the instance number 49,629 (the largest instance that Moped-*post** solved) *dual** is already 11 times faster than Moped-*post**. With the harder instances *dual** performs increasingly better than both *pre** and *post**.

Topology	Query	CEGAR	*dual**	Speedup
Colt	$\langle \cdot^* \rangle\ [\cdot\#\mathit{Toulouse}]\ [\hat{}\ \cdot\#\mathit{Milan}, \cdot\#\mathit{Poit}]^*\ [\mathit{Bari}\#\cdot]\ \langle \cdot^* \rangle\ 0$	0.94	90.54	96.42
Pern	$\langle \cdot^* \rangle\ [\cdot\#\mathit{N56}]\ [\hat{}\ \cdot\#\mathit{N38}, \cdot\#\mathit{Isla}, \cdot\#\mathit{N54}]^*\ [\mathit{N99}\#\cdot]\ \langle \cdot^* \rangle\ 0$	0.35	34.30	97.10
Colt	$\langle \cdot \rangle\ [\cdot\#\mathit{Paris}]\ \cdot^*\ [\mathit{Livorno}\#\cdot]\ \langle \cdot^+ \cdot \rangle\ 0$	1.00	>100.00	>100.00
Colt	$\langle \cdot ? \rangle\ [\cdot\#\mathit{Strasbourg}]\ \cdot^*\ [\cdot\#\mathit{Piacenza}]\ \cdot^*\ [\mathit{Novara}\#\cdot]\ \langle \cdot ? \rangle\ 0$	1.00	>100.00	>100.00
Colt	$\langle \cdot ? \rangle\ [\cdot\#\mathit{Karlsruhe}]\ \cdot^*\ [\cdot\#\mathit{Ostend}]\ \cdot^*\ [\mathit{Brindisi}\#\cdot]\ \langle \cdot ? \rangle\ 0$	0.98	>100.00	>102.04

Fig. 3: The queries that perform relatively best for CEGAR (time in seconds)

The performance of the CEGAR approach is incomparable with *dual** as on 27% of all instances CEGAR is faster (sometimes even by two orders of magnitude) but on the remaining instances it can be significantly slower. We noticed that the CEGAR approach is considerably better performing on negative queries (without any trace) where it is faster on 47% cases. The best cases for CEGAR with two orders of magnitude speedup are listed in Figure 3 and we remark that CEGAR solved 249 queries where *dual** timed out. The number of CEGAR iterations where the method is faster than *dual** ranges between 1 to 61 but typically less than 10 iterations are required to get a conclusive answer. As *dual** and CEGAR are incomparable, we use the pragmatic approach where we can run both of them in parallel and terminate as soon as one of the methods provides an answer. This is depicted by the curve min{*dual**, CEGAR} that further improves the performance by additional 20–30%. In particular this combined method is 7.5 times faster than AalWiNes-*post** on the median case and 17 times faster than Moped-*post** on the instance number 49,629.

Finally, as both the network encoding in AalWiNes [14] as well as in our paper overapproximate the set of network traces, they can provide inconclusive answers. On our benchmark, AalWiNes-*post** returned 2,024 inconclusive answers, whereas our encoding approach reported only 7 inconclusive answers for *dual** and 6 inconclusive answers for *dual** combined with CEGAR.

6 Conclusion

While more automated approaches to verify and operate communication networks can significantly improve their dependability, this requires efficient algorithms which can deal with the large scale and complexity of today's networks. We presented an efficient translation from MPLS routing tables into pushdown systems. We also revisited the problem of fast reachability analysis of pushdown systems and presented three techniques improving the performance over the state-of-the-art solution by an order of magnitude. In the future work we plan to study fast algorithms for verifying quantitative reachability properties (related to latency or network congestion) via weighted pushdown automata.

Acknowledgements. We thank to Bernhard Schrenk for updating the AalWiNes online demo at https://demo.aalwines.cs.aau.dk with the improved verification engine described in this paper.

References

1. Anderson, C.J., Foster, N., Guha, A., Jeannin, J.B., Kozen, D., Schlesinger, C., Walker, D.: NetKAT: Semantic foundations for networks. In: POPL'14. p. 113–126. ACM (2014)
2. Beckett, R., Mahajan, R., Millstein, T., Padhye, J., Walker, D.: Don't mind the gap: Bridging network-wide objectives and device-level configurations. In: ACM SIGCOMM'16. pp. 328–341. ACM (2016)
3. Bouajjani, A., Esparza, J., Maler, O.: Reachability analysis of pushdown automata: Application to model-checking. In: CONCUR'97. LNCS, vol. 1243, pp. 135–150. Springer (1997)
4. Büchi, J.R.: Regular canonical systems. Archiv für mathematische Logik und Grundlagenforschung **6**(3-4), 91–111 (1964)
5. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: CAV'00. LNCS, vol. 1855, pp. 154–169. Springer (2000)
6. Clarke, E.M., Grumberg, O., Long, D.E.: Model checking and abstraction. ACM Trans. on Prog. Lang. and Syst. **16**(5), 1512–1542 (Sep 1994)
7. Conway, C.L., Namjoshi, K.S., Dams, D., Edwards, S.A.: Incremental algorithms for inter-procedural analysis of safety properties. In: CAV'05. LNCS, vol. 3576, pp. 449–461. Springer (2005)
8. El-Hassany, A., Tsankov, P., Vanbever, L., Vechev, M.: Network-wide configuration synthesis. In: CAV'17. LNCS, vol. 10427, pp. 261–281. Springer (2017)
9. Esparza, J., Kiefer, S., Schwoon, S.: Abstraction refinement with Craig interpolation and symbolic pushdown systems. Journal on Satisfiability, Boolean Modeling and Computation **5**(1-4), 27–56 (2009)
10. Esparza, J., Knoop, J.: An automata-theoretic approach to interprocedural dataflow analysis. In: FOSSACS'99. LNCS, vol. 1578, pp. 14–30. Springer (1999)
11. Esparza, J., Schwoon, S.: A BDD-based model checker for recursive programs. In: CAV'01. LNCS, vol. 2102, pp. 324–336. Springer (2001)
12. Finkel, A., Willems, B., Wolper, P.: A direct symbolic approach to model checking pushdown systems. In: INFINITY'97. ENTCS, vol. 9, pp. 27–37. Elsevier (1997)
13. Jensen, J.S., Krøgh, T.B., Madsen, J.S., Schmid, S., Srba, J., Thorgersen, M.T.: P-Rex: Fast verification of MPLS networks with multiple link failures. In: CoNEXT. p. 217–227. ACM (2018)
14. Jensen, P.G., Kristiansen, D., Schmid, S., Schou, M.K., Schrenk, B.C., Srba, J.: AalWiNes: A fast and quantitative what-if analysis tool for MPLS networks. In: CoNEXT'20. p. 474–481. ACM (2020)
15. Kazemian, P., Varghese, G., McKeown, N.: Header space analysis: Static checking for networks. In: Proc. NSDI. pp. 113–126 (2012)
16. Knight, S., Nguyen, H., Falkner, N., Bowden, R., Roughan, M.: The internet topology Zoo. IEEE Journal on Selected Areas in Comm. **29**(9), 1765 –1775 (2011)
17. Pan, P., Swallow, G., Atlas, A.: Fast reroute extensions to RSVP-TE for LSP tunnels. RFC 4090, RFC Editor (2005), https://www.rfc-editor.org/rfc/rfc4090.txt
18. Schmid, S., Srba, J.: Polynomial-time what-if analysis for prefix-manipulating MPLS networks. In: IEEE INFOCOM'18. pp. 1799–1807. IEEE (2018)
19. Schwoon, S.: Model-checking pushdown systems. Ph.D. thesis, Technische Universität München (2002)
20. Schwoon, S.: Moped. In: http://www2.informatik.uni-stuttgart.de/fmi/szs/tools/moped/ (2002)
21. Suwimonteerabuth, D., Schwoon, S., Esparza, J.: jMoped: A java bytecode checker based on Moped. In: TACAS'05. LNCS, vol. 3440, pp. 541–545. Springer (2005)

Appendix

Proof Sketch for Proposition 1

We first prove the following lemma:

Lemma 1. *After initialization, the following invariant holds for any sequence of calls to* AddTransition *in Algorithm 1: for all $p \in P$ we have $(p, p) \xrightarrow{w}{}^{*} (q_1, q_2)$ iff $p \xrightarrow{w}{}^{*}_{1} q_1$ and $p \xrightarrow{w}{}^{*}_{2} q_2$, and we have $R = \{(q_1, q_2) \mid \exists p \in P.\ (p, p) \rightarrow^{*} (q_1, q_2)\}$.*

Proof (Sketch). Base: Initially $\rightarrow$ and R are empty. If $p \xrightarrow{w}{}^{*}_{1} q_1$ and $p \xrightarrow{w}{}^{*}_{2} q_2$, then the call AddState(p, p) will (using the recursive depth-first-search) add transitions to $\mathcal{A}_{\cap}$ such that $(p, p) \xrightarrow{w}{}^{*} (q_1, q_2)$, and add the state (q_1, q_2) to R (unless the initialization terminates early). No other transitions or states are added.

Invariant preservation: Let R, $\rightarrow$, $\rightarrow_1$ and $\rightarrow_2$ be the values before, and R', $\rightarrow'$, $\rightarrow'_1$ and $\rightarrow'_2$ be the values after a call to AddTransition($q_i \xrightarrow{\gamma}_i q'_i$). Consider each matching transition $q_{3-i} \xrightarrow{\gamma}_{3-i} q'_{3-i}$ in the other $\mathcal{P}$-automaton. If $(q_1, q_2) \in R$ then for some $p \in P, w \in \Gamma^{*}$ $(p, p) \xrightarrow{w}{}^{*} (q_1, q_2)$ and hence $p \xrightarrow{w}{}^{*}_{1} q_1$ and $p \xrightarrow{w}{}^{*}_{2} q_2$. Due to line 11 and 12 we have $p \xrightarrow{w\gamma}{}'^{*}_{1} q'_1$ and $p \xrightarrow{w\gamma}{}'^{*}_{2} q'_2$, and on line 13 we get $(p, p) \xrightarrow{w\gamma}{}'^{*} (q'_1, q'_2)$. Furthermore, all transitions and states reachable from (q'_1, q'_2) are added during the call to AddState(q'_1, q'_2) using depth-first-search of matching transitions in $\rightarrow'_1$ and $\rightarrow'_2$. If $(q_1, q_2) \notin R$ then for all $p \in P$ $(p, p) \not\rightarrow^{*} (q_1, q_2)$ and hence for all $w \in \Gamma^{*}$ either $p \overset{w}{\not\rightarrow}{}^{*}_{1} q_1$ or $p \overset{w}{\not\rightarrow}{}^{*}_{2} q_2$. Therefore either $p \overset{w\gamma}{\not\longrightarrow}{}'^{*}_{1} q'_1$ or $p \overset{w\gamma}{\not\longrightarrow}{}'^{*}_{2} q'_2$, and hence no transitions need to be added to $\mathcal{A}_{\cap}$ for this match. □

From Lemma 1 and the fact that the final states of $\mathcal{A}_{\cap}$ are $F_1 \times F_2$ we have that $Lang(\mathcal{A}_{\cap}) = Lang(\mathcal{A}'_1) \cap Lang(\mathcal{A}'_2)$.

From $R = \{(q_1, q_2) \mid \exists p \in P.\ (p, p) \rightarrow^{*} (q_1, q_2)\}$ as per Lemma 1, and the fact that line 6 of Algorithm 1, immediately after a state is added to R, checks whether it is a final state, we see that as soon as $Lang(\mathcal{A}_{\cap}) \neq \emptyset$, the algorithm returns true. This completes the proof of Proposition 1. □

Search Efficiency of CEGAR

The sets of possible link-header pairs, in particular C_0, can be very large, so we use two techniques to succinctly represent the search states. First, we use depth first search of the configuration sets $C_0 \underset{T_1}{\Longrightarrow} \ldots \underset{T_n}{\Longrightarrow} C_n$ to avoid storing most states in memory and to enable early termination if a valid reconstruction is found. We keep track of the current deepest C_i, for potentially computing refinement based on pushdown system rules. Second, we succinctly represent all headers in C_0 by a stack of wildcards with the correct size. When we follow a forwarding rule $t \in T_i$ during the depth first search, we specialize the wildcard to the required precondition label ℓ_i for that rule. We know the accepting path in

$\alpha(\mathcal{N}_a)$: $s_a \ldots \xrightarrow{\alpha(\ell_i)}_\alpha s_i \ldots$, so we check that in $\mathcal{N}_a$: $s_{i-1} \xrightarrow{\ell_i} s_i$ which eventually ensures that $h_0 \in \mathit{Lang}(\mathcal{N}_a)$. If there are still wildcards left, when we reach the final header h_n, we follow the remaining transitions from both $\mathcal{N}_a$ and $\mathcal{N}_c$ in lockstep to find concrete labels such that both $h_0 \in \mathit{Lang}(\mathcal{N}_a)$ and $h_n \in \mathit{Lang}(\mathcal{N}_c)$ are satisfied. If this is not possible, we have a spurious counter-example and find a refinement based on this. Finally, the search keeps track of the used and failed links, and avoids searching down branches, where it is already clear that the final check $\{e_0, \ldots, e_n\} \cap F = \emptyset$ and $|F| \leq k$ will fail.

Computing a Small Pair Refinement

For the CEGAR approach, we need a way of computing a refinement based on a spurious counter-example. The refinement should remove this spurious counter-example, while not making the next α-abstraction too fine-grained, since that would increase the size of PDS. The case where we have pairs $(\alpha(e), \alpha(\ell)) = (\alpha(e'), \alpha(\ell'))$ but $(e, \ell) \neq (e', \ell')$ turns out to be non-trivial. Here we abstract away from the use in CEGAR and define the problem of computing a small pair refinement as follows.

Given sets A and B, and sets of pairs $X \subseteq A \times B$ and $Y \subseteq A \times B$, find partitionings $A_1 \uplus \cdots \uplus A_n = A$ and $B_1 \uplus \cdots \uplus B_m = B$ with a) minimal n and m, such that b) for all i, j, $1 \leq i \leq n$, $1 \leq j \leq m$ we have $X \cap (A_i \times B_j) = \emptyset \vee Y \cap (A_i \times B_j) = \emptyset$.

We wish to avoid any i, j with $X \cap (A_i \times B_j) \neq \emptyset$ and $Y \cap (A_i \times B_j) \neq \emptyset$, since that would (locally) allow the spurious counter-example to reappear in the next iteration of CEGAR. Secondly, we wish to minimize n and m since this will keep the refinement as small as possible.

Since performance is important for this application, we present a greedy algorithm that satisfies b), while it relaxes a) to only small, rather than minimal, values for n and m.

We introduce the following notation: Functions $X_A(b) = \{a \mid (a, b) \in X\}$, $X_B(a) = \{b \mid (a, b) \in X\}$, $Y_A(b) = \{a \mid (a, b) \in Y\}$, $Y_B(a) = \{b \mid (a, b) \in Y\}$. Sets $X_A = \bigcup_{b \in B} X_A(b)$, $X_B = \bigcup_{a \in A} X_B(a)$, $Y_A = \bigcup_{b \in B} Y_A(b)$, $Y_B = \bigcup_{a \in A} Y_B(a)$.

Algorithm 3 assigns elements of A and B into buckets in a greedy manner, choosing first buckets for elements of A and then B in a way that ensures condition b) is fulfilled. When needed a new bucket is created.

Details on Tool for Generation of MPLS Data Plane

While the topologies of many real communication networks have been made available online, e.g., [16], this data does not include the router tables required to model MPLS data planes. For this paper, we hence develop a tool which allows to generate, for a given network topology, realistic synthetic data planes. Concretely, given a network topology (with weighted links as expected from an IGP topology), our tool directly computes the MPLS data plane that will be obtained after running typically deployed MPLS protocols Label Distribution

Algorithm 3 Greedy algorithm for computing pair refinement

```
 1: Initialize bucket $A_1$ with $A \setminus (X_A \cup Y_A)$
 2: for each $a$ in $X_A \cup Y_A$ do
 3:     for each existing bucket $A_i$ do
 4:         Let $Z_X \leftarrow \bigcup_{a' \in A_i} X_B(a')$, and $Z_Y \leftarrow \bigcup_{a' \in A_i} Y_B(a')$
 5:         if $(X_B(a) \cap Z_Y) \cup (Y_B(a) \cap Z_X) = \emptyset$ then
 6:             Put $a$ in bucket $A_i$
 7:     if $a$ was not assigned to a bucket then
 8:         Initialize new bucket with $a$
 9: Initialize bucket $B_1$ with $B \setminus (X_B \cup Y_B)$
10: for each $b$ in $X_B \cup Y_B$ do
11:     for each existing bucket $B_i$ do
12:         Let $Z_X \leftarrow \bigcup_{b' \in B_i} X_A(b')$, and $Z_Y \leftarrow \bigcup_{b' \in B_i} Y_A(b')$
13:         if $\forall (a, a') \in (X_A(b) \times Z_Y) \cup (Y_A(b) \times Z_X)$, $a$ and $a'$ not in same bucket then
14:             Put $b$ in bucket $B_i$
15:     if $b$ was not assigned to a bucket then
16:         Initialize new bucket with $b$
17: return all buckets $A_1 \dots A_n$ and $B_1 \dots B_m$
```

Protocol (LDP)[3] and Resource Reservation Protocol with Traffic Engineering extensions (RSVP-TE)[4] until convergence. The tool also allows instantiating industry-standard MPLS VPN services. The protocol related parameters can be adjusted for each experiment. By generating the forwarding tables according to usual protocols and practice, the result is a data plane for experimentation that shares similarities on its construction to the ones found on real MPLS networks.

Motivation for Redundant Paths and Labels on MPLS Networks

MPLS networks often feature significant redundancy in paths and labels, which can be exploited for optimization. There are a few reasons that explain why many paths on a MPLS network may have significant overlap or path redundancy. On networks that use LDP, this protocol is in charge of distributing and setting up paths across the network to reach IP prefixes present in routing tables of different routers. A typical strategy is to allocate a single MPLS label to all prefixes that can be reached through the same BGP next-hop[5]. Thus if the LDP process of a router at the edge of the MPLS domain creates a path tree for each of its BGP next-hops (a common situation when connecting with other networks), then the result is a network containing several labels along exactly the same paths, hence the redundancy. Also, LDP can be configured to introduce further deaggregation resulting in further redundancy[6] Another possibility is

[3] See https://www.rfc-editor.org/rfc/rfc5036.txt.
[4] See https://www.rfc-editor.org/rfc/rfc3209.txt.
[5] Juniper LDP overview https://www.juniper.net/documentation/us/en/software/junos/mpls/topics/topic-map/ldp-overview.html
[6] MPLS LDP FEC deaggregation https://www.juniper.net/documentation/en_US/junose15.1/topics/task/configuration/mpls-ldp-fec-deaggregation.html

due to having a few RSVP tunnel tailends concentrating many tunnels from different headends. In this case, a concentration of paths tend to appear when getting closer to the destination. This is not rare in practice for MPLS domain edge nodes connecting to IXPs or datacenters. The result is having many labels pointing to the same interface, or even more, to the same paths or common segments of paths, introducing a redundancy in the forwarding. Yet another situation on which path redundancy may arise is due to usage of One-to-One MPLS Fast Re Route protections [17]. In this case, given a RSVP path across the network, each router on said path computes an alternative path to forward packets in the event of link failure upstream, eventually merging with the original path. This protection might effectively multiply the number of forwarding paths in the network by a factor proportional to the average path length, increasing path redundancy if the ratio of tunnels to paths in the network is already high.