Eagle: Vulnerability and Congestion Aware Software Update Synthesis in Softwarized Networks

With a 5G Network Case Study

Nicolas Schnepf*, Remi Badonnel*, Damien Saucez†, Stefan Schmid‡ and Jiří Srba§
*Université de Lorraine, CNRS, Inria, Loria, France
†INRIA, Université Cote d'Azur, Sophia Antipolis, France
‡Technical University Berlin, Berlin, Germany §Aalborg University, Aalborg, Denmark

***Abstract*—Effective scheduling of software updates is a significant challenge in network operations and management, particularly when considering specific performance and security requirements. This paper focuses on the synthesis of such software updates in the context of emerging virtualized and softwarized networks, such as 5G network infrastructures, with the objective of ensuring vulnerability avoidance and congestion freedom at any time during the updates. We formalize the update synthesis problem and propose an algorithmic solution, called *Eagle*, that exploits formal methods and mixed integer linear programming, to achieve optimal solutions. We then complement it with a greedy algorithm to support faster computation. We exemplify our framework considering an implementation of a 5G architecture, as the one described in the ETSI 5123 standard, and which relies on kubernetes. Finally, we evaluate our approach through a large range of realistic ISP topologies from the *Topology Zoo* dataset, and we also perform extensive experiments on our kubernetes cluster, where we execute the software update sequences generated by our tool. This allows us to discuss the scalability of our approach along with its practical applicability.**

***Index Terms*—Update synthesis, Configuration and vulnerability management, Congestion avoidance, Softwarized networks**

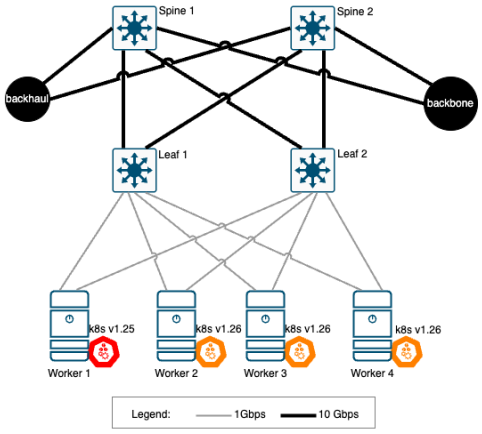


Fig. 1: 5G topology: the core is deployed in a spine-leaf topology where core functions are instantiated in a kubernetes cluster. Nodes in the cluster run either kubernetes version 1.25 or 1.26 and must be updated to version 1.27.

I. Introduction

The development of next generation networks is characterized by the growing softwarization of their resources, in order to offer further flexibility and adaptation to changes [1]. This softwarization paradigm allows us to automatically re-optimize and update networks at runtime. However, as network adaptions become more frequent and to ensure a high availability and dependability, it is crucial to maintain performance and safety properties also during updates, in a transient manner. From a security viewpoint, softwarized networks may also lead to additional vulnerabilities, that may increase their exposure to attacks. This introduces new challenges to human network operators. A network configuration may be vulnerable if it contains a certain combination of software versions and parameters. Fixing such vulnerabilities requires update procedures that are aware of compatibility constraints by, e.g., querying vulnerability descriptions, such as those that can be specified with the OVAL standard language [2]. The resulting update schedule must ensure that also during the updates, no vulnerability (violation of compatibility) is encountered, and that the system remains performant avoiding congestion.

As our main contribution, we jointly consider and solve the *vulnerability-aware* and *congestion-aware* software update synthesis problem by using a formal method approach which allows us to synthesize provably correct update schedules; while minimizing the number of update batches (thus shortening the overall update duration) and optimizing the size of each batch (to preserve network robustness). Our approach is generic and extensible, and we choose to illustrate our contribution with the concrete case of 5G networks [3].

A. 5G motivating example

Before presenting the problem and its model, let us overview our motivating example, a 5G case study. As there is a full functional decomposition in 5G networks with well defined interfaces between functions, it is possible to implement a 5G network with micro-services. As a consequence, it is now common practice to implement 5G networks using containers and deploy them in clusters within private clouds [4]. These compute infrastructures typically employ variations of a spine-leaf topology, as shown in Figure 1, which provides a high level of symmetry to support both high bandwidth and

robustness. We can idealise a 5G network as a set of Radio Access Networks (RANs) where each User Equipment (UE) is connected and that includes all the radio-related components, and a core network to which the RANs are connected. The core manages the infrastructure and forwards traffic to/from the RANs via the backhaul (black node on the left in Figure 1). We shall focus on the user plane in the core network. It is implemented with the User Plane Function (UPF) that transfers user traffic between the UEs and the Internet according to network policies [5]. For resiliency and performance reasons, the UPF is replicated on all worker nodes of the cluster. The UPF is a container, and the containers are coordinated in the cluster with kubernetes (k8s), which dynamically schedules the load on the worker nodes to accommodate to node failures. In addition, worker nodes must get the latest version of kubernetes. However, nodes get offline during an update, we thus need to find update sequences that guarantee that the cluster can support the load despite the reduction of available resources. Fewer resources however decrease robustness to failures. This is why the sequence should be as short as possible and at the same time grouping nodes to create equal-size batches of updates. Kubernetes adds another constraint: all nodes of a cluster must be within one minor version [6]. For instance, version 1.26 is compatible with version 1.25 and version 1.27, but version 1.25 is not compatible with version 1.27. To highlight this constraint, we have a mix of versions that all have to be updated to version 1.27 in a safe way (avoiding vulnerability) in our example.

B. Our contribution

We initiate the study of how to update a softwarized communication network in a provably correct manner, transiently accounting for software vulnerabilities (e.g., related to software compatibility) and congestion given the actual load in the network. After formalizing the problem, we propose *Eagle*, an algorithmic solution which relies on formal methods and linear programming, supporting an easy, automated and fast synthesis of correct updates. We complement it with a greedy algorithm to improve its performance. We are not aware of any existing update synthesis approach in the networking literature which combines both logical (vulnerability constraints) and quantitative (congestion) properties. We demonstrate, as a running example, our approach in the context of 5G architecture [7]. We further provide an analysis and empirical evaluation, assessing our solution through series of experiments, both using synthetic as well as real-world instances. We find that *Eagle* performs well and scales to realistic network sizes.

II. Formalization of Network Update Synthesis

We provide here the formalization of the vulnerability- and congestion-free network update problem.

Definition 1 (Network)**.** *A* network *is a directed multi-graph* $N = \langle V, E, src_E, dst_E, C, WP\rangle$ *consisting of a finite set of* nodes V*, a finite set of* links *(*edges*)* E*, a source function* $src_E : E \mapsto V$ *assigning a source node to each edge, a target function* $dst_E : E \mapsto V$ *assigning a target node to each edge, an edge capacity function* $C : E \mapsto \mathbb{N}$ *assigning a capacity to each edge and a set of waypoints* $WP \subseteq V$*.*

In a network there is a certain number of flow demands to satisfy; these demands represent the amount of traffic to route from one ingress node to another egress node in the network, while making sure that the traffic goes at least through one of the waypoint nodes. In the rest of this section we consider a fixed network $N = \langle V, E, src_E, dst_E, C, WP\rangle$.

Definition 2 (Demand)**.** *A* demand set *is a tuple* $F = \langle D, size, src_D, dst_D\rangle$ *where* D *is a finite set of demand names,* $size : D \mapsto \mathbb{N}$ *is a demand size function assigning a certain amount of traffic to each demand,* $src_D : D \mapsto V$ *is a demand source mapping function assigning a source node to each demand, and* $dst_D : D \mapsto V$ *is a demand target mapping function assigning a target node to each demand.*

We say that network N *has* sufficient capacity *to transfer the demands from* D*, if there is a multi-commodity flow for each demand that visits at least one of the waypoints* WP *while respecting the capacity constraints on the links.*

In our problem, we assume that each node $v \in V$ has its own software configuration given by a list of installed software components with different software versions.

Definition 3 (Software Configuration)**.** *Let* S *be a finite set of software systems and for each* $s \in S$ *let* $Dom(s)$ *be the set of possible versions of software* s*. We assume a natural comparison* $\leq$ *(older than) on the set* $Dom(s)$*. For each* $s \in S$*, the set* $Dom(s)$ *contains the symbol* $\perp$ *(incomparable with the other software versions) meaning that the software* s *is not installed. A* network configuration *is a function*

$$\delta : V \times S \to \cup_{s \in S} Dom(s)$$

s.t. $\delta(v, s) \in Dom(s)$ *for all* $v \in V$ *and* $s \in S$*, i.e.* $\delta(v, s)$ *returns the version of the software* s *installed on the node* v*.*

We say that a network configuration δ is *vulnerable* if it contains a certain combination of software configurations described, e.g., by OVAL vulnerability descriptions [2] and its extension to distributed vulnerabilities [8]. The OVAL specification is a Boolean combination of atomic predicates over the statements about the state (e.g. version number, parameters) of the software components installed on a node.

In general, a vulnerability formula ψ is an arbitrary Boolean combination of local node properties of the form:

$$\exists x.\varphi$$

where φ is a Boolean combination of atomic predicates like $x.s \bowtie state$ where $s \in S$, $\bowtie \in \{\leq, =, \geq\}$ and $state \in Dom(s)$. We say that the vulnerability ψ *manifests* in a configuration δ of a network N if for every node property $\exists x.\varphi$ in ψ there is a node v such that if we substitute $x.s \bowtie state$ with $\delta(v, s) \bowtie state$ in all atomic predicates inside of the node properties, the whole formula evaluates to true.

A typical example of a vulnerability formula from [9], [8] involving two nodes is:

$$\exists x.\ (x.\text{kamailio} \leq 5.6.4\ \wedge\ x.\text{pike} = \bot)\ \wedge$$
$$\exists x.\ (x.\text{bind} \geq 9.19.13 \wedge\ x.\text{external\_recursion} = \text{enabled})$$

that manifests in a software configuration where there is a node that has installed an older version (less than or equal to 5.6.4) of SIP server kamailio without the security module pike and at the same time there is a (possibly different) node running DNS bind server with a newer software version of at least 9.19.13 with external recursion enabled.

Another interesting example in the context of our 5G core topology is a violation of compatibility caused by concurrently running software versions of kubernetes that are more than one version from each other [6], which for example for the available software version of 1.24, 1.25, 1.26, and 1.27 is expressed by the vulnerability formula:

$$\big(\exists x.(x.\text{kubernetes} \leq 1.24)\ \wedge\ \exists x.(x.\text{kubernetes} \geq 1.26)\big)\ \vee$$
$$\big(\exists x.(x.\text{kubernetes} \leq 1.25)\ \wedge\ \exists x.(x.\text{kubernetes} \geq 1.27)\big)\ .$$

Software dependencies where a sofware s_1 depends on another software s_2 can be expressed by the vulnerability $\exists x.(\neg(x.s_1 = \bot) \wedge (x.s_2 = \bot))$ which manifests itsself whenever there is a node where software s_1 is installed but s_2 is missing. Similarly, we can define vulnerability predicates for some particular sofware version dependencies.

Several vulnerability constraints can be grouped into a single vulnerability using a conjunction. We shall now define the vulnerability- and congestion-free update problem.

Definition 4 (Update Synthesis Problem)**.** *Let N be a network, ψ a given network vulnerability formula, and δ_0 and δ_F the* initial *and* final *software configurations, respectively. The* update synthesis problem *is to find a sequence of update batches (disjoint set of nodes) $X_1, \ldots, X_k \subseteq V$ where $X_1 \cup \ldots \cup X_k = V$ such that for every i, $0 \leq i \leq k$, the intermediate software configuration δ_i (after finishing the updates in the first i batches) defined by*

$$\delta_i(v,s) = \begin{cases} \delta_F(v,s) & \text{if } v \in \bigcup_{j=1}^{i} X_j \\ \delta_0(v,s) & \text{otherwise} \end{cases}$$

does not manifest the network vulnerability ψ and the network N after removing the nodes X_i and their connected edges has a sufficient capacity to transfer all the traffic demands.

The intuition is that, given the update schedule $X_1, \ldots, X_k$, we shall, in the increasing order, take all the routers from the batch X_i offline, update their software from the initial to the final version and then bring them online. After this, we continue the process with X_{i+1}. Once all batches are updated, the network is in the final configuration δ_F. At any moment during the update, we may not encounter any vulnerability and the network without the routers X_i that are temporarily offline must have enough capacity to satisfy all traffic demands.

In practice, our first minimization objective is the number of batches that are required to complete the update as this will minimize the overall update time of the network; then our second minimization objective is to minimize the maximum batch size to preserve as much as possible network robustness.

III. Eagle: Linear Programming Algorithms

Our approach, *Eagle*, relies on formal methods and translates the update synthesis problem into a mixed integer linear program. We first introduce the necessary notation and then present the translation.

A. Preliminaries

A *mixed integer linear program* over the set of variables $x_1, \ldots, x_n$ is a finite set of linear constraints of the form $a_1x_1 + \ldots + a_nx_n \leq b$ where $a_1, \ldots, a_n$ and b are integer (or rational) numbers. The optimization problem is to find a *solution* (assignment of variables to either integers or rational numbers) that satisfies all constraints and minimizes/maximizes a given linear function $c_1x_1 + \ldots + c_nx_n$ where $c_1, \ldots, c_n$ are integer (or rational) constants. There exist numerous efficient tools for solving the NP-complete optimization problem for mixed integer linear programs [10].

We shall now proceed to formulate our update synthesis problem as a mixed integer linear program. Let $N = \langle V, E, src_E, dst_E, C, WP\rangle$ be a network, $F = \langle D, size, src_D, dst_D\rangle$ be a set of demands, S be a set of available software with corresponding versions $Dom(s)$ for all $s \in S$, let ψ be a vulnerability formula, and let δ_0 and δ_F be respectively the initial and final network configurations.

B. Exact approach for computing optimal solutions

An exact solution of our congestion-aware software update problem can be obtained by solving the mixed integer linear program from Figure 2 that relies on $B = \{b_1, \ldots, b_n\}$, a set of potential labels for update batches with n being the maximal number of such batches. In this program we use the following variables:

- $u : B \mapsto \{0,1\}$ where $u(b) = 1$ whenever the batch b is used in the update sequence,
- s maximal batch size,
- $x : B \times V \mapsto \{0,1\}$ where $x(b,v) = 1$ for all batches $b \in B$ where the vertex $v \in V$ is already updated,
- $y : B \times V \mapsto \{0,1\}$ where $y(b_i,v) = 1$ if and only if $x(b_{i-1},v) = 1$, and
- $\rho_1^d, \rho_2^d : B \times E \mapsto [0,1]$ where $\rho_1^d(b,e)$ is the portion of the traffic from $src_D(d)$ to the waypoints and $\rho_2^d(b,e)$ from the waypoints to $dst_D(d)$ for the demand d that is routed through the edge e in the batch b. Note that the ρ-variables are the only non-integer ones, which means that we allow to split flows along several paths.

Our goal is to minimize the objective function (1) that minimizes as the first priority the length of the update sequence (because of its $|V|$ coefficient) and as the second priority the maximal number of nodes per batch as stated in Constraint (19). Constraints (2), (3) and (4) impose that all variables can have a value of either 0 or 1, except for the ρ variables that take values from the interval $[0,1]$. Constraints (5) and (6) postulate that the ρ_1 flow starts in the source node of each demand and does not deliver anything to the demand's destination node as the waypoint(s) must be visited first and the full demand must be delivered to the waypoint node(s) as

$$\text{minimize: } |V| \cdot \sum_{b \in B} u(b) + s \qquad \text{subject to} \tag{1}$$

$$u(b) \in \{0,1\} \qquad \text{for } b \in B \tag{2}$$

$$x(b,v), y(b,v) \in \{0,1\} \qquad \text{for } b \in B, v \in V, e \in E \tag{3}$$

$$\rho_1^d(b,e), \rho_2^d(b,e) \in [0,1] \qquad \text{for } b \in B, v \in V, e \in E \tag{4}$$

$$\sum_{e \in E, src_E(e) = src_D(d)} \rho_1^d(b,e) = 1 \qquad \text{for } b \in B, d \in D \tag{5}$$

$$\sum_{e \in E, dst_E(e) = src_D(d)} \rho_1^d(b,e) = 0 \qquad \text{for } b \in B, d \in D \tag{6}$$

$$\sum_{e \in E, dst_E(e) \in WP} \rho_1^d(b,e) = 1 \qquad \text{for } b \in B, d \in D \tag{7}$$

$$\sum_{e \in E, src_E(e) \in WP} \rho_1^d(b,e) = 0 \qquad \text{for } b \in B, d \in D \tag{8}$$

$$\sum_{e \in E, dst_E(e) = dst_D(d)} \rho_2^d(b,e) = 1 \qquad \text{for } b \in B, d \in D \tag{9}$$

$$\sum_{e \in E, src_E(e) = dst_D(d)} \rho_2^d(b,e) = 0 \qquad \text{for } b \in B, d \in D \tag{10}$$

$$\sum_{e \in E, dst_E(e) = w} \rho_1^d(b,e) = \sum_{e \in E, src_E(e) = w} \rho_2^d(b,e) \qquad \text{for } d \in D, b \in B, w \in WP \tag{11}$$

$$\sum_{e \in E, dst_E(e) = v} \rho_i^d(b,e) = \sum_{e \in E, src_E(e) = v} \rho_i^d(b,e) \qquad \text{for } i \in \{1,2\}, b \in B, d \in D, v \in V, \; v \notin WP \cup \{src_D(d), dst_D(d)\} \tag{12}$$

$$y(b_i, v) = x(b_{i-1}, v) \qquad \text{for } i \in \{2, \ldots, n\}, b_i, b_{i-1} \in B, v \in V \tag{13}$$

$$\sum_{d \in D} (\rho_1^d(b,e) + \rho_2^d(b,e)) \cdot size(d) \leq C(e) \cdot (1 - \big(x(b,v) - y(b,v)\big)) \qquad \text{for } b \in B, e \in E, v \in \{src_E(e), dst_E(e)\} \tag{14}$$

$$x(b_i, v) \geq x(b_{i-1}, v) \qquad \text{for } i \in \{2, \ldots, n\}, b_i, b_{i-1} \in B, v \in V \tag{15}$$

$$|V| \cdot u(b) \geq \sum_{v \in V} x(b,v) - y(b,v) \qquad \text{for } b \in \{b_2, \ldots, b_n\} \tag{16}$$

$$\sum_{b \in B} x(b,v) - y(b,v) = 1 \qquad \text{for } v \in V \tag{17}$$

$$u(b_i) \leq u(b_{i-1}) \qquad \text{for } i \in \{2, \ldots n\} \tag{18}$$

$$s \geq \sum_{v \in V} x(b,v) - y(b,v) \qquad \text{for } b \in B \tag{19}$$

Fig. 2: Linear program for exact solutions

specified by Constraint (7) while no flow from the waypoints returns back to ρ_1 by Constraint (8). Similarly, Constraints (9) and (10) require that the ρ_2 flow delivers the full demand size to its destination while the source nodes of the demand do not contribute to this flow. This is because ρ_2 receives all its traffic exclusively from the waypoint node(s) as specified by Constraint (11). Constraint (12) imposes the classical flow preservation property (sum of incoming flow is equal to the sum of the outgoing flow) for both ρ_1 and ρ_2 and for all nodes, except the source, destination and the waypoints of the demand. Constraint (13) states that y variables are set to 1 one batch latter then their x equivalent; hence the expression $x(b,v) - y(b,v)$ is 1 only for the batch b where the node v is updated. Constraint (14) requires that the flow forwarded through an edge cannot exceed its capacity and that no traffic can pass through an edge connected to a node that is currently being updated. Constraint (15) states that once a node is updated in a batch it remains updated in the next batches. Constraint (16) guarantees that whenever a node is updated in a batch then this batch is counted in the length of the update sequence and Constraint (17) states that a node can be updated only once. Finally, Constraint (18) ensures that a batch is used in the update sequence if and only if its direct predecessor is also used (hence avoiding gaps in update batches).

$$red(x(b,v)=1) = \begin{cases} c = x(b,v) \\ \text{where } c \in \{0,1\} \text{ is a fresh variable} \end{cases}$$

$$red(x(b,v)=0) = \begin{cases} c = 1 - x(b,v) \\ \text{where } c \in \{0,1\} \text{ is a fresh variable} \end{cases}$$

$$red(\neg\varphi) = \begin{cases} c = 1 - c' \\ \text{where } c \in \{0,1\} \text{ is a fresh variable} \\ \text{and the variable } c' \text{ is declared in } red(\varphi) \end{cases}$$

$$red(\bigvee_{1\le i\le m} \varphi_i) = \begin{cases} m \cdot c \ge \sum_{1\le i\le m} c_i \\ c \le \sum_{1\le i\le m} c_i \\ \text{where } c \in \{0,1\} \text{ is a fresh variable} \\ \text{and variables } c_i \text{ are declared in } red(\varphi_i) \end{cases}$$

$$red(\bigwedge_{1\le i\le m} \varphi_i) = \begin{cases} c \ge \sum_{1\le i\le m} c_i - (m-1) \\ c \le c_i \\ \text{where } c \in \{0,1\} \text{ is a fresh variable} \\ \text{and variables } c_i \text{ are declared in } red(\varphi_i) \end{cases}$$

Fig. 3: Constraints for vulnerability (for each $b \in B$)

Remark 1. *If the set of waypoints is empty, we modify the constraints by removing the ρ_2 variables and for each demand set the waypoint equal to the destination of the demand.*

In addition to constraints from Figure 2 that guarantee congestion-freedom, we also add constraints from Figure 3 that enforce that no vulnerability constraints in the formula ψ are broken. Recall that the vulnerability predicate ψ manifests in a given network configuration δ if and only if for every node property $\exists x.\varphi$ there exists a node $v \in V$ that make φ is satisfied. Notice that due to the construction so far, any given batch $b \in B$ determines a software configuration δ where for every $v \in V$ holds that $\delta(v) = \delta_0(v)$ iff $x(b,v) = 0$, and $\delta(v) = \delta_F(v)$ iff $x(b,v) = 1$.

We can now replace in ψ any occurence of $\exists x.\varphi$ with $\bigvee_{v\in V}(\varphi)$ and inside of φ we replace any atomic test asking if there is a node v with a software version of the software component s satisfying the corresponding software version in a batch $b \in B$ of the form $x.s \bowtie state$ by one of the following expressions:

- $x(b,v) = 0 \ \vee \ x(b,v) = 1$ if $\delta_0(v,s) \bowtie state$ and $\delta_F(v,s) \bowtie state$
- $x(b,v) = 0$ if $\delta_0(v,s) \bowtie state$ and $\delta_F(v,s) \not\bowtie state$
- $x(b,v) = 1$ if $\delta_0(v,s) \not\bowtie state$ and $\delta_F(v,s) \bowtie state$
- *false* otherwise.

This converts the distributed vulnerability formula ψ into a Boolean formula over the atomic predicates of the form $x(b,v) = 0$ or $x(b,v) = 1$. Finally, we convert this Boolean formula into a set of linear constraints by applying the reduce operator *red* as detailed in Figure 3.

Here we inductively construct constraints for each subformula by introducing a fresh integer variable $c \in \{0,1\}$ where 0 means that the subformula is false and 1 stands for true. The first two introduce a fresh subformula for the atomic predicates with the expected meaning. To encode the negation $\neg\varphi$, we apply the third reduction rule and create a new variable c that we set equal to $1-c'$ where c' is the variable for the subformula φ. This effectively implements the negation operator. The fourth rule for disjunction adds two constraints. The first one requires that if at least one of the c_i variables is 1 (implying that the subformula φ_i is true) then c must be set to 1 as well. The second constraint for disjunction enforces that c is 0 if all subformulae are false, i.e. $\sum_{1\le i\le m} c_i = 0$. The last rule for conjunction also adds two constraints. The first one says that if all subformulate are true, meaning that $\sum_{1\le i\le m} c_i = m$, then c must be 1 as well. The second constraint says that if at least one subformula is false then c must be 0 too. Finally, assuming that c is the variable declared for the whole formula ψ, we add the constraint $c = 0$ that requires that the vulnerability ψ is false (i.e. it does not manifest in any of the batches $b \in B$).

The combination of the constraints from Figures 2 and 3 enforces that a solution to the linear program (i.e. an assignment of nodes to update batches) does not cause any congestion nor vulnerability. Moreover, the linear program returns a solution that minimizes the number of batches as the primary criterion and the maximum batch size as a secondary one.

C. Greedy approach for computing close-to-optimal solutions

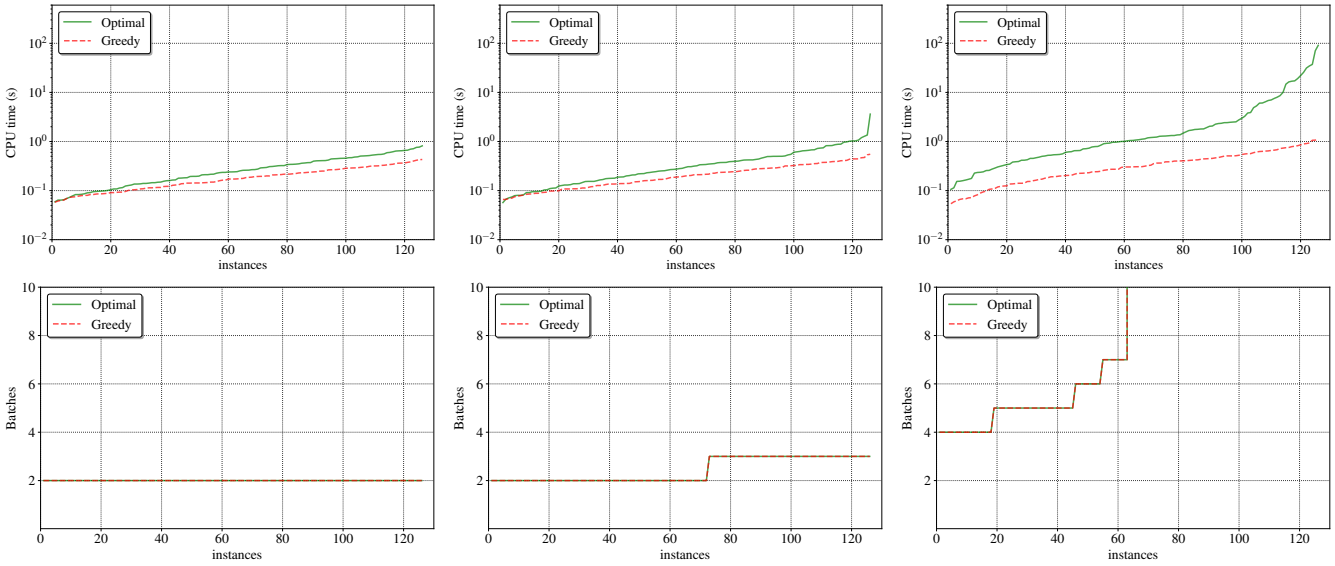
Computing the optimal update sequence using our mixed integer linear program above can be computationally expensive because for each batch $b \in B$ we create a new copy of all variables. Hence *Eagle* also uses a faster greedy approach which computes one batch at a time only but does not in general guarantee to find an update sequence with the minimum number of batches. Our greedy synthesis approach is derived by modifying the linear program above by:

- setting the batch sequence length to one ($B = \{b_1\}$),
- maximizing the objective function $\sum_{v\in V} x(b_1,v)$,
- limiting the maximal batch size in a dedicated constraint;
- removing Constraints (13) and (15) to (19), and
- replacing $y(b_i,v)$ by $y(b_1,v)$ in Constraint (14).

By doing so we obtain a linear program that maximizes the number of nodes that can be at the moment safely updated in the network. We implement those updates and repeat the procedure, again greedily maximizing the number of nodes that can be updated concurrently, until all nodes get updated. It is not guaranteed that this algorithm returns the shortest update sequence but our practical experiments show that in many cases this is indeed the case, at the exchange for a significantly faster synthesis of update batches.

IV. Experimental Evaluation

We implemented our optimal and greedy algorithms for computing batches of update sequences in Python 3 with more than 1500 lines of code. We use GLPK version 5.0 We evaluate our approach on two different benchmarks, as well as on a physical test-bed network described in Figure 1. First, we consider the *5G spine-leaf* reference topology [5] as depicted



(a) Low load (b) Medium load (c) High load

Fig. 4: Computation time and batch length for 5G spine-leaf networks under different loads.

in Figure 1—we scale this topology to include from 4 to 10 worker nodes and 2 to 10 leaf and spine nodes. For the flow demand between the backbone and the backhaul, we consider three different scenarios with demands that correspond to 25%, 50%, and 75% of the total network capacity. We refer to them as low, medium, and high load scenarios. Second, we consider a benchmark from the *Topology Zoo* [11] with gravity model demands and $WP = \emptyset$. For each network in either of the benchmarks, we randomly select 60% of nodes that have to be updated and we consider two vulnerability scenarios: (i) the kubernetes scenario (for the 5G benchmark as well as the ISP topologies) where 10% of selected nodes run kubernetes 1.25, all other nodes run kubernetes 1.26 and all nodes move to kubernetes 1.27; (ii) the kamailio scenario (for the ISP topologies) where 10% of selected nodes run kamailio without pike installed and have to install it and all other nodes run a DNS server, version 9.19.12 and have to move to version 9.19.13. These two vulnerabilities are formalized in Section II.

All experiments are run on a cluster with 16 nodes, all of them having 2 CPUs Intel Xeon Gold 5218R with 20 cores and 96GB of RAM running SMP Debian 5.10.179-2. The timeout for each experiment is set to 10 minutes. Updating a network is two-fold process. First, the update sequence must be computed and then the actual network must be updated. In Section IV-A we evaluate the time needed to compute optimal sequence of updates and compare it with the greedy approach, also with respect to the number of computed batches. In these experiments we do not evaluate the maximal batch size since it is only a secondary objective that is only optimized once the shortest update is obtained. In Sections IV-B we then measure the actual update time on a physical network delivering 5G connectivity running kubernetes and compare the traditional (sequential) update approach with the one computed by us.

A. Simulation benchmarks

Figure 4 shows the CPU time required to solve the mixed integer linear programs for the 5G spine-leaf benchmark, as well as the number of computed batches. The instances on the x-axis are sorted by the increasing CPU time (resp. number of batches) shown on the y-axis. We can observe that in any load scenario, our greedy algorithm computes the shortest number of batches (the same number as the exact optimal algorithm) and is fast enough to be considered at runtime as it returns the batches in less than one second. On the other hand, the complexity of computing the optimal solution using the exact algorithm grows with the number of batches (the y-axis is logarithmic) that need to be considered and for the high load scenario it can take up to 100 seconds in the worst scenarios. As we may expect, the higher the load we have in the network, the more batches are required to execute the updates without causing congestion. In fact, for the high load scenario, only 63 instances were updatable (using up to 7 batches) and for the remaining scenarios there is no safe update sequence. In the practical application, this means that we cannot update from kubernetes version 1.25 directly to 1.27 but we have to split the updates by first moving to version 1.26 and then to 1.27.

Figure 5 summarizes the results for the ISP networks from the *Topology Zoo*, where for each topology we consider 30 largest elephant flows with three different size of demands and we include both the kamailio and kubernetes vulnerability predicates. We only depict the problem instances where the optimal approach finished within 10 minutes. We can clearly see that the CPU time to compute the update batches using the greedy approach is fast (under 30 seconds in the worst-case instances) while the optimal algorithm requires considerably more time to synthetize the optimal update batches. In this benchmark, which is not as symmetric as the 5G spine-leaf

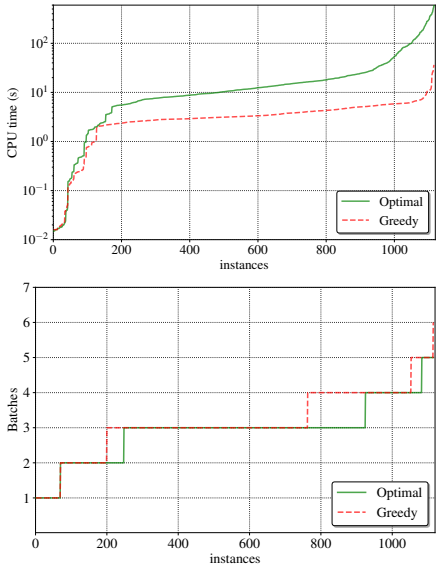


Fig. 5: Computation time and batch length for *Topology Zoo*.

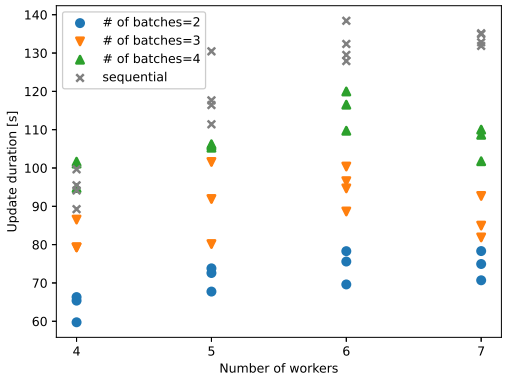


Fig. 6: Update duration on the cluster. Worker nodes are updated to kubernetes v1.27, one node initially runs version 1.25 and the remaining ones run 1.26. Optimal update sequences with 2 to 4 batches are compared to the sequential one.

topologies, it is not the case anymore that the greedy approach always computes the optimal solution and in 19.6% of cases it returns an update sequence with one additional batch than necessary. Only in a single case the greedy approach returns a solution that requires two additional batches compared to the optimal one. We can hence conclude that our greedy approach returns in a matter of seconds close-to-optimal number of update batches and making it practically applicable.

B. Measurements in our kubernetes cluster

We shall now focus on practical implementation of software updates in our kubernetes cluster in order to measure how much update time can be gained in a real system. We consider the 5G spine-leaf cluster in Figure 1 with 4 to 7 worker nodes. The cluster runs a 5G *User Plane Function* (UPF), which role is to forward traffic between user equipment and the Internet. We implement the function with a kubernetes *deployment* where the pod anti-affinity scheduling policy is used and where the number of replicas is set to the number of worker nodes. This guarantees that one and only one UPF instance runs on every single healthy worker node, at any time. The UPF is accessed from the outside (i.e., from the backhaul or backbone) thanks to the kubernetes *LoadBalancer service* that transparently spreads the load between active instances of the UPF, ensuring so continuity of service. We determine the optimal worker node update sequences with *Eagle* such that there is enough UPF function capacity to carry UE/Internet traffic, even during updates, and that kubernetes version skew is respected during the updates. To demonstrate the possible gains of our method, we compare the update sequence time with the default sequence, which consists in updating the workers one by one. We implement this procedure with Ansible by using the SLICES post-5G blueprint and respect the standard kubernetes cluster upgrade recommendations [12]. Using the sequence generated by *Eagle* to actually update the cluster is straightforward. The output of the algorithm is a sequence of batches, each batch defines all the nodes to be updated in this batch. This information is translated in an Ansible inventory on which the playbook performing the update is called. The Ansible playbook doing the update is decomposed in three phases. First, it isolates the nodes to update from the cluster (with *kubectl drain*), then it updates the software version of the nodes. Finally, it uncordons (with *kubectl uncordon*) the nodes so that they reintegrate the cluster. Given the definition of the deployment of the UPF, kubernetes automatically schedules back the UPF on the updated nodes. This is sequentially repeated for each batch.

The cluster is implemented with virtual machines running on a Linux Rocky 9.1rt server offered by the SLICES research infrastructure with an Intel Xeon Gold 6254 CPU and 64GB of RAM. Each worker node is a Linux Ubuntu 20.04 KVM virtual machine with 2 vCPU and 4GB of RAM.

Figure 6 gives the update duration in seconds for different number of workers (from 4 to 7) and different number of update batches. The x-axis shows the number of workers and on the y-axis shows the measured update duration. Each measurement is repeated three times. As in Section IV-A, we consider three cases: (i) low load which updates in only two batches (as computed by our algorithm); (ii) medium load which requires three batches (again computed by our algorithm); and (iii) high load, resulting in a need of four batches to fully update the cluster (as computed by our algorithm). The load on the network does not influence the update duration but it constraints the number of batches in the sequence. As updates within a batch are done in parallel on the nodes in the batch, it is the number of batches itself that drives the update duration, not the number of nodes. This is confirmed by Fig. 6 where update duration for a given number of batches is similar regardless of the number of nodes. We can observe a rather linear increase of update duration in the sequential approach as one node is updated after the other. For

four worker nodes the sequential approach and updates with four batches exhibit similar duration as the sequences are the same. As one node in the cluster is one version lower than the other, this node always needs two updates. Hence the first batch takes the time of two version updates while the other batches take the time of only one.

We can conclude that the update duration is not bound to the number of workers but to the number of update batches, justifying the need to minimize the number of batches. Interestingly, updating nodes in the cluster in safe way with our algorithm is fast, taking only a few seconds to compute the optimal update sequence. In practice, we can so keep clusters up-to-date and secure without service interruption.

V. Related work

Open Vulnerability and Assessment Language (OVAL), part of the SCAP protocol [13], standardizes the description of vulnerabilities as a logical combination of conditions to be observed on the configuration of a system [2]. Some approaches have also investigated the specification of distributed vulnerabilities that may be spread over several nodes in a network, such as the DOVAL language defined on top of OVAL [14]. The remediation activities and their automation to bring the system back to a secure state is crucial in order to efficiently minimize the attack surface of the system over time. In this regard, methods for evaluating the risks associated with changes, such as proposed in [15], provide a support for vulnerability management, particularly for taking decisions about effective change implementations. Solutions such as [16] demonstrate the benefits of verification techniques in order to prevent new vulnerabilities when corrective operations are executed. While some of these remediation approaches address distributed vulnerabilities, they do not consider the update process in itself, and even less its impact on congestion.

Approaches to consistently update network configurations have recently been studied in the networking community, primarily motivated by the advent of software-defined networks but also more generally [17]. Existing work on consistent network update problems can roughly be classified into two categories: problems in which a certain logical property (e.g., loop freedom) must be ensured during reconfiguration and problems in which a certain quantitative property (e.g., congestion freedom) must be ensured. Our work contributes with a first scalable approach which accounts for both dimensions at the same time. Regarding the logical properties, the seminal work by Reitblatt et al. [18] considers a problem in which a strong logical property needs to be preserved, *per-packet consistency*, which is well-aligned with the strong consistency properties usually provided in traditional networks [19]. Mahajan and Wattenhofer [20] started exploring the benefits of relaxing the per-packet consistency property, while *transiently* providing only essential logical properties like loop-freedom. The authors also present a first algorithm that quickly updates routes in a transiently loop-free manner, and their study was recently refined in [21], [22], [23], where the authors also establish hardness results, as well as in [24], [25], [26], [27], which respectively, focus on the problem of minimizing the number of scheduling rounds [26], initiate the study of multiple policies [24], and introduce additional transient routing constraints related to waypointing [25], [27]. However, we none of these papers considers bandwidth capacity constraints.

Research efforts on network update problems taking into account quantitative properties have also been conducted in the past, especially considering congestion issues which affect performance. Hongqiang et al. [28] presented a congestion-free rerouting solution which has already been extended in several papers, using static [29], [30], [31], [32], dynamic [33], or time-based [34], [35] approaches. Vissicchio et al. presented FLIP [36], which combines per-packet consistent updates with order-based rule replacements, to reduce memory overhead. Amiri et al. [37] presented a polynomial-time algorithm for acyclic flow graphs, to compute feasible (possibly very long) update schedules. In follow-up work, Amiri et al. [38] presented a polynomial-time algorithm that computes the fastest update schedule for two flows, and also showed that the problem is already NP-hard for six flows. Even though there are several solutions to the network update problem using formal methods and automated approaches, also for the synthesis [39], [40], [41], [42], [43], [44], none of these approaches support simultaneously both logical and quantitative approaches, which is the main contribution of our work.

VI. Conclusion

We initiated a study on optimal update software synthesis in a computer network in a consistent and vulnerability-aware manner, taking into account also possible congestion during the update process. We presented an automated approach for synthesizing update schedules automatically and with the smallest possible number of update batches—an aspect important for decreasing the overall duration of the complete network update. We provided two algorithms relying on this approach, the first one computing the shortest possible update sequence, the second one greedily computing each update batch as the largest possible set of remaining nodes to update. We compared these two approaches, and demonstrated their practical applicability in the context of 5G networks, as well as on a large benchmark of ISP Internet topologies. An extensive empirical evaluation shows that our approach is tractable and scales to realistic network sizes.

As a future work it would be interesting to establish a standardized database of distributed network vulnerabilies that could be automatically translated to our framework and extend the routing properties from basic waypointing to more sophisticated ones like 5G service chaining.

Acknowledgments

This work has been partially supported by the French National Research Agency under the France 2030 label (NF-HiSec ANR-22-PEFT-0009, NF-Plateforme XG ANR-22-PEFT-00011), and by the German Federal Ministry of Education and Research (BMBF), grant 16KISK020K (6G-RIC).

References

[1] W. Kellerer, P. Kalmbach, A. Blenk, A. Basta, M. Reisslein, and S. Schmid, "Adaptable and Data-driven Softwarized Networks: Review, Opportunities, and Challenges," *Proceedings of the IEEE*, vol. 107, no. 4, pp. 711–731, 2019.

[2] MITRE Corporation, "OVAL: Open Vulnerabilities Assessment Language." https://oval.mitre.org. Last visited on June 2024.

[3] N. Al-Falahy and O. Y. Alani, "Technologies for 5G Networks: Challenges and Opportunities," *It Professional*, vol. 19, no. 1, 2017.

[4] N. Nikaein, M. K. Marina, S. Manickam, A. Dawson, R. Knopp, and C. Bonnet, "OpenAirInterface: A Flexible Platform for 5G Research," *SIGCOMM Comput. Commun. Rev.*, vol. 44, p. 33–38, oct 2014.

[5] L. Peterson and O. Sunay, *5G Mobile Networks: A Systems Approach*. Springer International Publishing, 2020.

[6] "Version Skew Policy — kubernetes.io." https://kubernetes.io/releases/version-skew-policy/. [Accessed 26-10-2023].

[7] T. ETSI, "123 501 v16. 6.0 (oct. 2020)," *Technical Specification G*, vol. 5.

[8] M. Barrère, R. Badonnel, and O. Festor, "Towards the assessment of distributed vulnerabilities in autonomic networks and systems," in *Proc. of the IEEE/IFIP Network Operations and Management Symposium*, pp. 335–342, 2012.

[9] G. Zhang, S. Ehlert, T. Magedanz, and D. Sisalem, "Denial of Service Attack and Prevention on SIP VoIP Infrastructures Using DNS Flooding," in *Proc. of the 1st International Conference on Principles, Systems and Applications of IP Telecommunications*, IPTComm, (New York, NY, USA), p. 57–66, Association for Computing Machinery, 2007.

[10] R. Kannan and C. L. Monma, "On the computational complexity of integer programming problems," in *Optimization and Operations Research* (R. Henn, B. Korte, and W. Oettli, eds.), (Berlin, Heidelberg), pp. 161–172, Springer Berlin Heidelberg, 1978.

[11] S. Knight, H. Nguyen, N. Falkner, R. Bowden, and M. Roughan, "The internet topology zoo," *Selected Areas in Communications, IEEE Journal on*, vol. 29, pp. 1765 –1775, october 2011.

[12] "Upgrading Kubeadm Clusters." https://kubernetes.io/docs/tasks/administer-cluster/kubeadm/kubeadm-upgrade. Last Visited on May 2024.

[13] J. Banghart and C. Johnson, "The Technical Specification for the Security Content Automation Protocol (SCAP). NIST Special Publication." https://csrc.nist.gov/Projects/Security-Content-Automation-Protocol/SCAP-Releases, 2011. Last visited on June 2023.

[14] M. Barrère, R. Badonnel, and O. Festor, "Towards the Assessment of Distributed Vulnerabilities in Autonomic Networks and Systems," in *Proc. of the IEEE/IFIP Network Operations and Management Symposium (IEEE/IFIP NOMS)*, pp. 335–342, Apr. 2012.

[15] J. Sauve, R. Santos, R. Reboucas, and A. Moura, "Change Priority Determination in IT Service Management Based on Risk Exposure," *IEEE Transactions on Network and Service Management*, vol. 5, pp. 178–187, Sept. 2008.

[16] M. Barrére, R. Badonnel, and O. Festor, "A SAT-based Autonomous Strategy for Security Vulnerability Management," in *2014 IEEE Network Operations and Management Symposium (NOMS)*, pp. 1–9, 2014.

[17] K.-T. Foerster, S. Schmid, and S. Vissicchio, "Survey of Consistent Network Updates," in *ArXiv Technical Report*, 2016.

[18] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker, "Abstractions for Network Update," in *Proc. of ACM SIGCOMM*, 2012.

[19] P. Cerný, N. Foster, N. Jagnik, and J. McClurg, "Optimal consistent network updates in polynomial time," in *Distributed Computing - 30th International Symposium, DISC 2016, Paris, France, September 27-29, 2016. Proceedings*, pp. 114–128, 2016.

[20] R. Mahajan and R. Wattenhofer, "On Consistent Updates in Software Defined Networks," in *Proc. ACM HotNets*, 2013.

[21] S. A. Amiri, A. Ludwig, J. Marcinkowski, and S. Schmid, "Transiently consistent sdn updates: Being greedy is hard," in *23rd International Colloquium on Structural Information and Communication Complexity, SIROCCO*, 2016.

[22] K.-T. Förster, R. Mahajan, and R. Wattenhofer, "Consistent Updates in Software Defined Networks: On Dependencies, Loop Freedom, and Blackholes," in *Proc. 15th IFIP Networking*, 2016.

[23] K.-T. Förster and R. Wattenhofer, "The Power of Two in Consistent Network Updates: Hard Loop Freedom, Easy Flow Migration," in *Proc. 25th International Conference on Computer Communication and Networks (ICCCN)*, 2016.

[24] S. Dudycz, A. Ludwig, and S. Schmid, "Can't Touch This: Consistent Network Updates for Multiple Policies," in *Proc. IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2016.

[25] A. Ludwig, S. Dudycz, M. Rost, and S. Schmid, "Transiently Secure Network Updates," in *Proc. of ACM SIGMETRICS*, 2016.

[26] A. Ludwig, J. Marcinkowski, and S. Schmid, "Scheduling loop-free network updates: It's good to relax!," in *in Proc. of ACM PODC*, 2015.

[27] A. Ludwig, M. Rost, D. Foucard, and S. Schmid, "Good Network Updates for Bad Packets: Waypoint Enforcement Beyond Destination-Based Routing Policies," in *Proc. ACM Workshop on Hot Topics in Networks (HotNets)*, 2014.

[28] H. H. Liu, X. Wu, M. Zhang, L. Yuan, R. Wattenhofer, and D. A. Maltz, "zUpdate: Updating Data Center Networks with Zero Loss," in *Proc. ACM SIGCOMM*, 2013.

[29] S. Brandt, K.-T. Förster, and R. Wattenhofer, "On Consistent Migration of Flows in SDNs," in *Proc. 36th IEEE International Conference on Computer Communications (INFOCOM)*, 2016.

[30] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer, "Achieving High Utilization with Software-driven WAN," in *ACM SIGCOMM Computer Communication Review*, vol. 43, pp. 15–26, 2013.

[31] M. Reitblatt, N. Foster, J. Rexford, and D. Walker, "Consistent Updates for Software-defined Networks: Change You Can Believe In!," in *Proc. ACM HotNets*, 2011.

[32] J. Zheng, H. Xu, G. Chen, and H. Dai, "Minimizing Transient Congestion During Network Update in Data Centers," in *Proc. 23rd IEEE International Conference on Network Protocols (ICNP)*, 2015.

[33] X. Jin et al., "Dynamic Scheduling of Network Updates," in *Proc. of ACM SIGCOMM*, 2014.

[34] T. Mizrahi and Y. Moses, "Time-based Updates in Software Defined Networks," in *Proc. ACM HotSDN*, pp. 163–164, 2013.

[35] T. Mizrahi, O. Rottenstreich, and Y. Moses, "TimeFlip: Scheduling Network Updates with Timestamp-based TCAM Ranges," in *Proc. IEEE INFOCOM*, pp. 2551–2559, 2015.

[36] S. Vissicchio and L. Cittadini, "FLIP the (Flow) Table: Fast LIghtweight Policy-preserving SDN Updates," in *Proc. IEEE INFOCOM*, 2016.

[37] S. Akhoondian Amiri, S. Dudycz, S. Schmid, and S. Wiederrecht, "Congestion-Free Rerouting of Flows on DAGs," in *Proc. of International Colloquium on Automata, Languages, and Programming (ICALP)*, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

[38] S. Amiri, S. Dudycz, M. Parham, S. Schmid, and S. Wiederrech, "On Polynomial-Time Congestion-Free Software-Defined Network Updates," in *Proc. of the IFIP International Networking Conference*, 2019.

[39] N. Christensen, M. Glavind, S. Schmid, and J. Srba, "Latte: Improving the Latency of Transiently Consistent Network Update Schedules," *ACM SIGMETRICS Perf. Evaluation Review*, vol. 48, no. 3, pp. 14–26, 2021.

[40] M. Didriksen, P. G. Jensen, J. F. Jønler, A.-I. Katona, S. D. Lama, F. B. Lottrup, S. Shajarat, and J. Srba, "Automatic Synthesis of Transiently Correct Network Updates via Petri Games," in *Proc. of the International Conference on Petri Nets*, Springer, 2021.

[41] K. G. Larsen, A. Mariegaard, S. Schmid, and J. Srba, "AllSynth: Transiently Correct Network Update Synthesis Accounting for Operator Preferences," in *Theoretical Aspects of Software Engineering: 16th International Symposium, TASE 2022, Cluj-Napoca, Romania, July 8–10, 2022, Proceedings*, pp. 344–362, Springer, 2022.

[42] A. El-Hassany, P. Tsankov, L. Vanbever, and M. Vechev, "Network-Wide Configuration Synthesis," in *Computer Aided Verification: 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II 30*, pp. 261–281, Springer, 2017.

[43] A. El-Hassany, P. Tsankov, L. Vanbever, and M. Vechev, "Netcomplete: Practical Network-wide Configuration Synthesis with Autocompletion," in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pp. 579–594, 2018.

[44] S. Schmid, B. C. Schrenk, and A. Torralba, "NetStack: A Game Approach to Synthesizing Consistent Network Updates," in *Prof. of IFIP Networking Conference (IFIP Networking)*, pp. 1–9, IEEE, 2022.