Efficient Loop Detection in Forwarding Networks and Representing Atoms in a Field of Sets

Yacine Boufkhad[*]

Leonardo Linguaglossa[†]

Fabien Mathieu[‡]

Diego Perino[§]

Laurent Viennot[¶]

September 5, 2018

Abstract

The problem of detecting loops in a forwarding network is known to be NP-complete when general rules such as wildcard expressions are used. Yet, network analyzer tools such as Netplumber (Kazemian et al., NSDI'13) or Veriflow (Khurshid et al., NSDI'13) efficiently solve this problem in networks with thousands of forwarding rules. In this paper, we complement such experimental validation of practical heuristics with the first provably efficient algorithm in the context of general rules. Our main tool is a canonical representation of the atoms (i.e. the minimal non-empty sets) of the field of sets generated by a collection of sets. This tool is particularly suited when the intersection of two sets can be efficiently computed and represented. In the case of forwarding networks, each forwarding rule is associated with the set of packet headers it matches. The atoms then correspond to classes of headers with same behavior in the network. We propose an algorithm for atom computation and provide the first polynomial time algorithm for loop detection in terms of number of classes (which can be exponential in general). This contrasts with previous methods that can be exponential, even in simple cases with linear number of classes. Second, we introduce a notion of network dimension captured by the overlapping degree of forwarding rules. The values of this measure appear to be very low in practice and constant overlapping degree ensures polynomial number of header classes. Forwarding loop detection is thus polynomial in forwarding networks with constant overlapping degree.

Keywords: Forwarding tables, Loop, Software-defined networking, Field of sets

[*] Université Paris Diderot
[†] Inria
[‡] Nokia Bell Labs
[§] Telefonica Research
[¶] Inria

1 Introduction

With the multiplication of network protocols, network analysis has become an important and challenging task. We focus on a key diagnosis task: detecting possible forwarding loops. Given a network and node forwarding tables, the problem consists in testing whether there exists a packet header h and a directed cycle in the network topology such that a packet with header h will indefinitely loop along the cycle. This problem is indeed NP-complete as noted in [19]. Its hardness comes from the use of compact representations for predicate filters: the set of headers that match a rule is classically represented by a prefix in IP forwarding, a general wildcard expression in Software-Defined Networking (SDN), value ranges in firewall rules, or even a mix of such representations if several header fields are considered.

We first give a toy example of forwarding loop problem where the predicate filter of each rule is given by a wildcard expression, that is an ℓ-letter string in $\{1, 0, *\}^{\ell}$. Such an expression represents the set all ℓ-bit headers obtained by replacing each $*$ of the expression by either 0 or 1. A packet with header in that set is said to match the rule. Figure 1 illustrates a one node network with wildcard expressions of $\ell = 4$ letters. Rules are tested from top to bottom. All rules indicate to drop packets except the last one that forwards packets to the node itself. This network contains a forwarding loop if there exists a header $x_1x_2x_3x_4$ that matches no rule except the last one. Not matching a rule as $110*$ corresponds to having $x_1 = 0$, $x_2 = 0$, or $x_3 = 1$. This one node network thus has a forwarding loop iff the formula $(\overline{x_1} \vee \overline{x_2} \vee \overline{x_3} \vee \overline{x_4}) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_3} \vee x_4) \wedge (\overline{x_1} \vee \overline{x_2} \vee x_3) \wedge (\overline{x_1} \vee x_2) \wedge (x_1)$ is satisfiable, which is not the case. This simple example can easily be generalized to reduce SAT to forwarding loop detection in networks with wildcard rules. It also points out a key problem: testing the emptiness of expressions such as $r_p \setminus \cup_{i=1..p-1} r_i$ where $r_1, \ldots, r_p$ are the sets associated to p rules.

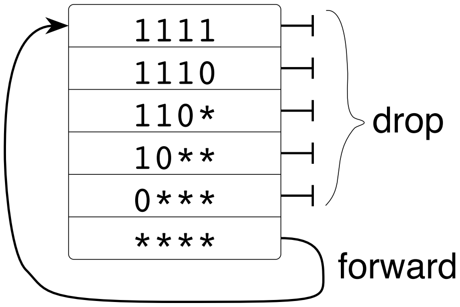

Figure 1: Does this one node network have a forwarding loop ?

As packet headers in practical networks such as Internet typically have hundreds of bits, search of the header space is completely out of reach. The main challenge for solving such a problem thus resides in limiting the number of tests to perform. For that purpose, previous works [17, 15] propose to consider sets of headers that match some predicate filters and do not match some others. Defining two headers as equivalent when they match exactly the same predicate filters, it then suffices to perform one test per equivalence class. These classes are indeed the atoms (the minimal non-empty sets) of the field of sets (the (finite) σ-algebra) generated by the sets associated to the rules.

A first challenge lies in efficiently identifying and representing these atoms. This would be fairly easy if both intersection and complement could be represented efficiently. In practice, most classical compact data-structures for sets of bit strings are closed under intersection but not under complement. For example, the intersection of two wildcard expressions, if not empty, can obviously be represented by a wildcard expression, but the complement of a wildcard expression is more problematic. Previous works overcome this difficulty by representing the complement of a ℓ-letter wildcard expression as the union of several wildcard expressions (up to ℓ). However, this can result in exponential blow-up and the tractability of these methods rely on various heuristics that do not offer rigorously proven guarantees.

A second challenge lies in understanding the tractability of practical networks. One can easily design a collection of 2ℓ wildcard expressions that generates all the 2^ℓ possible singletons as atoms (all the ℓ-letters strings with only one non-$*$ letter). What does prevent such phenomenon in practice? Can we provide a property that intuitively fits with practical network and guarantees that the number of atoms does not blow up? This paper aims at addressing both challenges with provable guarantees.

Related work The interest for network problem diagnosis has recently grown after the advent of Software-Defined Networking (SDN) [23, 18, 21, 12]. SDN offers the opportunity to manage the forwarding tables of a network with a centralized controller where full knowledge is available for analysis. Previous works has led to a series of methods for network analysis, resulting in several tools [17, 13, 19, 27]. The main approaches rely on computing classes of headers by combining rule predicate filters using intersection and set difference (that is intersection with complement). The idea of considering all header classes generated by the global collection of the sets associated to all forwarding rules in the network is due to Veriflow [17]. However, the use of set differences results in computing a refined partition of the atoms of the field of sets generated by this collection that can be much larger than an exact representation. NetPlumber [13], which relies on the header space analysis introduced in [15], refines this approach by considering the set of headers that can follow a given path of the network topology. This set is represented as a union of classes that match some rules (those that indicate to forward along the path) and not some others (those that have higher priority and deviate from the path): a similar problem of atom representation thus arises. The idea of avoiding complement operations is somehow approached in the optimization called "lazy subtraction" that consists in delaying as much as possible the computation of set differences. However, when a loop is detected, formal expressions with set differences have to be tested for emptiness. They are then actually developed, possibly resulting in the manipulation of expressions with exponentially many terms.

Concerning the tractability of the problem, the authors of NetPlumber observe a phenomenon called "linear fragmentation" [15, 14] that allows to argue for the efficiency of the method. They introduce a parameter c measuring this linear fragmentation and claim a polynomial time bound for loop detection for low c [15] (when emptiness tests are not included in the analysis). However, the rigorous analysis provided in [14] includes a c^{D_G} factor where D_G is the diameter of the network graph. While this factor appears to be largely overestimated in practice, the sole hypothesis of linear fragmentation does not suffice for explaining tractability and prove polynomial time guarantees. The alternative approach of Veriflow is specifically optimized for rules resulting from range matching within each field of the header. When the number of fields is constant, polynomial time execution can be guaranteed but this result does not extend to general wildcard matching.

A similar problem consists in conflict detection between rules and their resolution [1, 9, 5]. It has mainly been studied in the context of multi-range rules [1, 9], which can benefit from computational geometry algorithms. (A multi-range can be seen as a hyperrectangle in a d-dimensional euclidean space where d is the number of fields composing headers.) Another similar problem, determining efficiently the rule that applies to a given packet, has been studied for multi-ranges [9, 10, 11]. In the case of wildcard matching, such problems are related to the old problem of partial matching [24]. It is believed to suffer from the "curse of dimensionality" [3, 22] and no method significantly faster than exhaustive search is expected to be found with near linear memory (although some tradeoffs are known for small number of $*$ letters [7]). However, efficient hardware-based implementations exist based on Ternary Content Addressable Memory (TCAMs) [4] or Graphics Processing Unit (GPU) [26].

Regarding the manipulation of set collections, recent work [20] shows how to enumerate all sets obtained by closure under given set operations with polynomial delay. In particular, this allows to produce the field of sets generated by the collection. However, this setting requires a set representation which explicitly lists all elements and does not apply here. Another issue comes from the fact that the field set can be exponentially larger than its number of atoms.

Rule repr.	Trivial	NetPlumber [13]	Veriflow [17]	This paper
T_ℓ-bounded	$O(T_\ell n n_G 2^\ell)$	–	–	$O(T_\ell n m^2 + n n_G m)$
” ov. deg. $O(\log n)$	”	–	–	$T_\ell n^{O(1)} m + O(n_G m \log n)$
” ov. deg. k	”	–	–	$O((T_\ell n + T_\ell k^2 2^k \log n + k n_G) n^k)$
ℓ-wildcard	$O(\ell n n_G 2^\ell)$	$\Omega(\ell n_G 2^{\min(\ell,n)})$	$\Omega(n_G 2^{\min(\ell/2,n)})$	$O(\ell n m^2 + n n_G m)$
” ov. deg. k	”	$\Omega(\ell n_G 2^{\min(\ell,n)})$	$\Omega(n_G 2^{\min(\ell/2,n)})$	$O((\ell n + \ell k 2^k + k n_G) n^k)$
d-multi-rng.	$O(\ell n n_G (2n)^d)$	–	$\Omega((\frac{n}{d})^{d-1} n_G \frac{m}{d})$	$O(d n m^2 + n n_G m)$
” ov. deg. k	”	–	$\Omega((\frac{n}{d})^{d-1} n_G \frac{m}{d})$	$O((\ell k^{d+1} \log^d n + \ell k 2^k + k n_G) n^k)$

Table 1: Worst-case complexity of forwarding loop detection with n rules that generate m header classes in an n_G-node network, for various rule set representations: T_ℓ-bounded for intersection and cardinality computations in $O(T_\ell)$ time; ℓ-wildcard for wildcard expressions with ℓ letters; d-multi-rng. for multi-ranges in dimension d (with $\ell = O(d)$). Additional hypothesis “ov. deg. k” stands for overlapping degree of rule sets bounded by k.

Our contributions First, we make a key algorithmic step by providing an efficient algorithm for computing an exact representation of the atoms of the field of sets generated by a collection of sets. The representation obtained is linear in the number of atoms and allows to test efficiently if an atom is included in a given set of the collection. The main idea is to represent an atom by the intersection of the sets that contain it. We avoid complement computations by using cardinality computations for testing emptiness. Our algorithm is generic and supports any data-structure for representing sets of ℓ-bit strings that supports intersection and cardinality computation in bounded time $O(T_\ell)$ for some value T_ℓ. It runs in polynomial time with respect to n, m, the number of sets and atoms respectively. Beyond combinations of wildcard and range expressions, we believe that it could be extended to support expressions on hashed values (for a fixed hash function) or bloom filters by defining auxiliary cardinality measures.

Second, we provide a dimension parameter, the *overlapping degree* k, that captures the complexity of a collection of rule sets considered in a forwarding network. It is defined as the maximum number of distinct rules (i.e. with pairwise distinct associated sets) that match a given header. This parameter constitutes a measure of complexity for the field of sets generated by a given collection of sets. In the context of practical hierarchical networks, we have the following intuitive reason to believe that this parameter is low: in such networks, more specific rules are used at lower levels of the hierarchy. We can thus expect that the overlapping degree is bounded by the number of layers of the hierarchy. Empirically, we observed a value within $5-15$ for datasets with hundreds to thousands of distinct multi-field rules, and $k = 8$ for the collection of IPv4 prefixes advertised in BGP. A constant overlapping degree implies that the number of header classes is polynomially bounded, giving a hint on why practical networks are tractable despite the NP-completeness of the problem. In addition, the algorithm we propose is tailored to take advantage of low overlapping degree k, even without knowledge of k. Table 1 provides a summary of the complexity results obtained for loop detection depending on how the sets associated to rules are represented. Our techniques can be extended to handle write actions (partial writes of fixed values such as field replacement) while maintaining polynomial guarantees.

Our algorithm for atom computation solves two difficult technical issues. First, it manages to remain polynomial in the number m of atoms even though the number of sets generated by intersection solely can be exponential in m with general rules. Second, the use of cardinality computations allows to avoid exponential blow-up (in contrast with previous work) but naturally induces a quadratic $O(m^2)$ term in the complexity. However, we manage to reduce it to $O(m)$ in the case of collections with logarithmic overlapping degree (i.e. $k = O(\log n)$). Indeed, our algorithm then becomes linear in m and polynomial in n, providing an algorithmic breakthrough towards efficient atom enumeration.

Roadmap: Section 2 introduces the model. Section 3 describes how to represent the atoms of a field of sets. We state in Section 4 our main result concerning atom computation and its implications for forwarding loop detection that give the upper bounds listed in Table 1. Section 5 gives more insight about the comparison of our results with previous works and justifies the lower bounds presented in Table 1. Finally, Section 6 discusses some perspectives.

2 Model

2.1 Problem

We consider a general model of network where a *network instance* $\mathcal{N}$ is characterized by:

- a graph $G = (V, E)$ where each node $u \in V$ is a *router* and has a *forwarding table* $T(u)$.
- a natural number ℓ representing the (fixed) bit-length of packet headers.

Let H denote the set of all 2^ℓ possible headers (all ℓ-bit strings). Each forwarding table $T(u)$ is an ordered list of forwarding rules $(r_1, a_1), \dots, (r_p, a_p)$. Each *forwarding rule* (r, a), is made of a predicate filter r and an action a to apply on any packet whose header matches the predicate. We say that a header h *matches* rule (r, a) when it matches predicate r (we may equivalently say that (r, a) matches h). We then write $h \in r$ to emphasize the fact that r can be viewed as a compact data-structure encoding the set of headers that match it. This set is called the *rule set* associated to (r, a). For the ease of notation, we thus let r denote both the predicated filter of rule (r, a) and the associated set.

We consider three possible actions for a packet: forward to a neighbor, drop, or deliver (when the packet has reached destination). The priority of rules is given by their ordering: when a packet with header h arrives at node u, the first rule matched by h is applied. Equivalently, the rule (r_i, a_i) is applied when $h \in r_i \cap \overline{r_1} \cap \cdots \cap \overline{r_{i-1}}$, where $\overline{r}$ denotes the complement of r. When no match is found (i.e. $h \in \overline{r_1} \cap \cdots \cap \overline{r_p}$), the packet is dropped.

Given a header h, the *forwarding graph* $G_h = (V, E_h)$ of h represents the forwarding actions taken on a packet with header h: $uv \in E_h$ when the first rule that matches h in $T(u)$ indicates to forward to v. The *forwarding loop detection* problem consists in deciding whether there exists a header $h \in H$ such that G_h has a directed cycle.

Note that we make the simplifying assumption that the input port of an incoming packet is not taken into account in the forwarding decision of a node. In a more general setting, a node has a forwarding table for each incoming link. This is essentially the same model except that we consider the line-graph of G instead of G.

2.2 Header Classes

A natural relation of equivalency exists between headers with respect to rules: two headers are equivalent if they match exactly the same rules, that is if they belong to the same rule sets. Trivially, two equivalent headers let the corresponding packets have exactly the same behavior in the network. The resulting equivalence classes partitions the header set H into nonempty disjoint subsets called *header classes*. To check any property of the network, it suffices to do it on a class-by-class basis instead of a header-by-header basis. The number m of header classes is thus a natural parameter when considering the difficulty of forwarding loop detection (or other similar network analysis problems). A more accurate definition of classes could take into account the order of rules and the topology (see Appendix E) but this does not allows us to obtain complexity gain in general.

The header classes can be defined according to the collection $\mathcal{R}$ of rule sets of $\mathcal{N}$ (i.e. $\mathcal{R} = \{r \mid \exists u, a \text{ s.t. } (r, a) \in T(u)\}$). If $\mathcal{R}(h) \subseteq \mathcal{R}$ denotes the set of all rule sets associated to the rules matched by a given header h, then its header class is clearly equal to $\left(\cap_{r \in \mathcal{R}(h)} r\right) \cap \left(\cap_{r \in \mathcal{R} \setminus \mathcal{R}(h)} \overline{r}\right)$ (with the convention $\cap_{r \in \emptyset} r = H$). Such sets are the atoms of the field of sets generated by $\mathcal{R}$. Their computation is the main topic of this paper and is detailed in the next section.

2.3 Set representation

As we focus on the collection $\mathcal{R}$ of rule sets, we now detail our hypothesis on their representation. We assume that a data-structure $\mathcal{D}$ allows to represent some of the subsets of a space H. For the ease of notation, $\mathcal{D} \subseteq \mathcal{P}(H)$ also denotes the collection of subsets that can be represented with $\mathcal{D}$. We assume that $\mathcal{D}$ is closed under intersection: if s and s' are in $\mathcal{D}$, so is $s \cap s'$. We say that

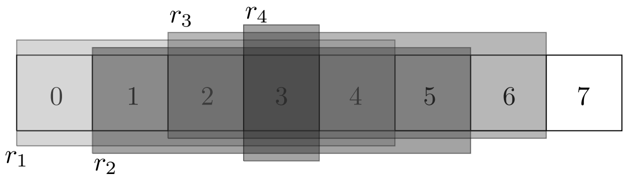

Figure 2: Toy example of an 8-elements space H with a collection $\mathcal{R} = \{r_1, r_2, r_3, r_4\}$.

such a data-structure $\mathcal{D}$ for subsets of H is T_H*-bounded* when intersection and cardinality can be computed in time T_H at most: given the representation of $s, s' \in \mathcal{D}$, the representation of $s \cap s' \in \mathcal{D}$ and the size $|s|$ of s (as a binary big integer) can be computed within time T_H. As big integers computed within time T_H have $O(T_H)$ bits, this implies $|H| = 2^{O(T_H)}$: the bound T_H obviously depends on H. Intersection, inclusion test ($s \subseteq s'$), cardinality computation ($|s|$) and cardinality operations (addition, subtraction and comparison) are called *elementary set operations*. Under the T_H-bounded hypothesis, all these operations can be performed in $O(T_H)$ time ($s \subseteq s'$ is equivalent to $|s \cap s'| = |s|$).

Two typical examples of data-structures meeting the above requirements are wildcard expressions and multi-ranges. In a forwarding network, we consider the header space $H = \{0,1\}^\ell$ of all ℓ-bit strings, which may be decomposed in several fields. A rule set is typically represented by a wildcard expression or a range of integers. In both cases, they can be represented within 2ℓ bits and both representation are $O(\ell)$-bounded. We call ℓ*-wildcard* a string $e_1 \cdots e_\ell \in \{0,1,*\}^\ell$. It represents the set $\{x_1 \cdots x_\ell \in \{0,1\}^\ell \mid \forall i, x_i = e_i \text{ or } e_i = *\}$. If rules are decomposed into fields, any combinations of wildcard expressions and ranges can be used (either one for each field) and represented within $O(\ell)$ bits. However cardinality computations can take $\Theta(\ell \log \ell)$ time as multiplications of big integers are required. Such representation is thus $O(\ell \log \ell)$-bounded. Given d field lengths $\ell_1, \ldots, \ell_d$ with sum ℓ, we call (d,ℓ)*-multi-range* a cartesian product $[a_1, b_1] \times \cdots \times [a_d, b_d]$ of d integer ranges with $0 \leq a_i \leq b_i < 2^{\ell_i}$ for i in $1..d$. It represents the set $\{bin(x_1, \ell_1) \cdots bin(x_d, \ell_d) \mid (x_1, \ldots, x_d) \in [a_1, b_1] \times \cdots \times [a_d, b_d]\}$ where $bin(x_i, \ell_i)$ is the binary representation of x_i within ℓ_i bits.

When manipulating a collection of p sets in $\mathcal{D}$, we assume that their representations are stored in a balanced binary search tree, allowing to dynamically add, remove or test membership of a set in $O(T_H \log p)$ time. More efficient data-structures (tries and segment trees) can be used for wildcard expressions and multi-ranges as detailed in Appendix A.1.

3 Atoms and combinations generated by a collection of sets

Given a space of elements H, a *collection* is a finite set of subsets of H. The *field of sets* $\sigma(\mathcal{R})$ generated by a collection $\mathcal{R}$ is the (finite) σ-algebra generated by $\mathcal{R}$, that is the smallest collection closed under intersection, union and complement that contains $\mathcal{R} \cup \{\emptyset, H\}$.

The *atoms* of $\sigma(\mathcal{R})$ are classically defined as the non-empty elements that are minimal for inclusion. For brevity, we call them the atoms generated by $\mathcal{R}$. Let $\mathcal{A}(\mathcal{R})$ denote their collection. Note that for $a \in \mathcal{A}(\mathcal{R})$ and $r \in \mathcal{R}$, we have either $a \subseteq r$ or $a \subseteq \overline{r}$ (otherwise $a \cap r$ and $a \cap \overline{r}$ would be non-empty elements of $\sigma(\mathcal{R})$ strictly included in a). This gives a characterization of the atoms that matches our definition of header classes when $\mathcal{R}$ is the collection of rule sets of a network (see Section 2.2):

$$\mathcal{A}(\mathcal{R}) = \left\{a \neq \emptyset \mid \exists R \subseteq \mathcal{R}, a = (\cap_{r \in R} r) \cap \left(\cap_{r \in \mathcal{R} \setminus R} \overline{r}\right)\right\}. \tag{1}$$

For example, the collection $\mathcal{R}$ pictured in Figure 2 generates 7 atoms: $\{0\} = r_1 \cap \bar{r_2} \cap \bar{r_3} \cap \bar{r_4}$, $\{1\} = r_1 \cap r_2 \cap \bar{r_3} \cap \bar{r_4}$, $\{2,4\} = r_1 \cap r_2 \cap r_3 \cap \bar{r_4}$, $\{3\} = r_1 \cap r_2 \cap r_3 \cap r_4$, $\{5\} = \bar{r_1} \cap r_2 \cap r_3 \cap \bar{r_4}$, $\{6\} = \bar{r_1} \cap \bar{r_2} \cap r_3 \cap \bar{r_4}$, and $\{7\} = \bar{r_1} \cap \bar{r_2} \cap \bar{r_3} \cap \bar{r_4}$.

Due to the *complement* operations, the atoms can be harder to represent than the rules they are generated from. In Figure 2, all rules are ranges, but the atom $\{2,4\}$ is not. When the *intersection* operation can be computed and represented efficiently (see Section 2.3), it is natural to consider the collection $\mathcal{C}(\mathcal{R}) \subseteq \sigma(\mathcal{R})$ of *combinations* defined as sets that can obtain by intersection from sets in $\mathcal{R}$:

$$\mathcal{C}(\mathcal{R}) := \{c \neq \emptyset \mid \exists R \subseteq \mathcal{R}, c = \cap_{r \in R} r\}\,. \tag{2}$$

In Figure 2, there are 8 combinations: r_1, $r_1 \cap r_2$, $r_1 \cap r_3$, r_4, $r_2 \cap r_3$, r_3, H and r_2.

Given a collection $\mathcal{R}$ and a subset $s \in H$, we let $\mathcal{R}(s) := \{r \in \mathcal{R} \mid s \subseteq r\}$ denote the *containers* of s, that is the sets in $\mathcal{R}$ that contain s. We associate each combination $c \in \mathcal{C}(\mathcal{R})$ with the set $a(c) := c \cap \left(\cap_{r \in \mathcal{R} \setminus \mathcal{R}(c)} \overline{r}\right)$. The function $a(\cdot)$ (with parenthesis) should not be confused with an atom a (without parenthesis). A combination c is said to be *covered* if $a(c) = \emptyset$ (the union of its non-containers covers it), otherwise it is *uncovered*. Similarly, we associate each atom $a \in \mathcal{A}(\mathcal{R})$ with the combination $c(a) := \cap_{r \in \mathcal{R}(a)} r$ (this corresponds to the "positive" part of the characterization from Equation 1). The following proposition states that atoms can be represented by uncovered combinations.

Proposition 1. The collection $\mathcal{UC}(\mathcal{R}) := \{c \in \mathcal{C}(\mathcal{R}) \mid a(c) \neq \emptyset\}$ of uncovered combinations is in one-to-one correspondence with the atom collection $\mathcal{A}(\mathcal{R})$: each $c \in \mathcal{UC}(\mathcal{R})$ corresponds to the atom $a(c)$. Reciprocally, each atom a is mapped to combination $c(a)$.

Based on Proposition 1, we say that an uncovered combination c *represents* atom $a(c)$. $\mathcal{UC}(\mathcal{R})$ can be seen as a canonical representation of $\mathcal{A}(\mathcal{R})$ by a collection of combinations. In Figure Figure 2, atom $\{2, 4\}$ is represented by combination $r_1 \cap r_3 = r_1 \cap r_2 \cap r_3$. Similarly, atoms $\{0\}$, $\{1\}$, $\{3\}$, $\{5\}$, $\{6\}$ and $\{7\}$ are represented by (uncovered) combinations r_1, $r_1 \cap r_2$, r_4, $r_2 \cap r_3$, r_3, and H respectively. Combination r_2 is covered: $a(r_2) = \bar{r_1} \cap r_2 \cap \bar{r_3} \cap \bar{r_4} = \emptyset$.

The proof of Proposition 1 is straightforward. First, we verify that if c is an uncovered combination, $a(c)$ is an atom: it suffices to observe that $c = \cap_{r \in \mathcal{R}(c)} r$ and to match $a(c) = c \cap \left(\cap_{r \in \mathcal{R} \setminus \mathcal{R}(c)} \overline{r}\right)$ with the atom characterization given by Equation 1 using $R = \mathcal{R}(c)$. Similarly, if $a = (\cap_{r \in R} r) \cap \left(\cap_{r \in \mathcal{R} \setminus R} \overline{r}\right)$ is an atom for some $R \subseteq \mathcal{R}$, the combination $c(a)$ satisfies $\mathcal{R}(c(a)) = R$ which implies $a(c(a)) = a$. In particular, as $a \neq \emptyset$, $c(a)$ is uncovered.

A nice property of the characterization of Proposition 1 is that it allows to efficiently test whether a set $r \in \mathcal{R}$ contains an atom $a \in \mathcal{A}(\mathcal{R})$: given a combination c that represents a, $a \subseteq r$ is equivalent to $c \subseteq r$. This comes from the fact that every uncovered combination c has same containers as $a(c)$. (If it was not the case, $a(c) = \emptyset$ and c is covered.) This explains the importance of determining $\mathcal{UC}(\mathcal{R})$, and in particular to separate covered combinations from uncovered ones. This is the subject of the next section, but we first formally introduce the notion of overlapping degree.

Overlapping degree of a collection Our representation is naturally associated to the following measure of complexity of a collection $\mathcal{R}$. We define the *overlapping degree* k of $\mathcal{R}$ as the maximum number of containers of an element, that is $k = \max_{h \in H} |\mathcal{R}(\{h\})|$. Note that all elements within an atom have same containers in $\mathcal{R}$ and that a set cannot have more containers than any of its elements. We thus have:

$$k = \max_{s \subseteq H} |\mathcal{R}(s)| = \max_{a \in \mathcal{A}(\mathcal{R})} |\mathcal{R}(a)|$$

As any atom a can be expressed as $a = (\cap_{r \in R} r) \cap \left(\cap_{r \in \mathcal{R} \setminus R} \overline{r}\right)$ where $R = \mathcal{R}(a)$ is the set of containers of a, the number of atoms is obviously bounded by $\binom{n}{k}$ where $n = |\mathcal{R}|$ denotes the number of sets in $\mathcal{R}$. The overlapping degree of a collection thus measures its complexity in terms of number of atoms it may generate.

We similarly define the *average overlapping degree* $\overline{k}$ as the average number of containers of an atom: $\overline{k} = \frac{\sum_{a \in \mathcal{A}(\mathcal{R})} |\mathcal{R}(a)|}{|\mathcal{A}(\mathcal{R})|}$. We obviously have $\overline{k} \leq k$. Given a collection $\mathcal{R}$, we will also consider the average overlapping degree $\overline{K}$ of combinations, that is the average overlapping degree of $\mathcal{C}(\mathcal{R})$. Note that $\mathcal{C}(\mathcal{R})$ has the same collection of atoms as $\mathcal{R}$ and that a combination $c = \cap_{r \in \mathcal{R}(c)} r$ containing an atom a must satisfy $\mathcal{R}(c) \subseteq \mathcal{R}(a)$. The overlapping degree of $\mathcal{C}(\mathcal{R})$ is thus at most 2^k and we always have $\overline{K} \leq 2^k$.

In the example from Figure 2, one can verify that we have $k = 4$, $\overline{k} = 13/7$ and $\overline{K} = 4$.

In real datasets we observe that both k and $\overline{K}$ are in the range $[2, 15]$ while $\overline{k}$ is in the range $[1.5, 5]$ (these include Inria firewall rules, Stanford forwarding tables provided by Kazemian et al. [16] and IPv4 prefixes announced at BGP level from Route Views [25]).

4 Incremental computation of atoms

We can now state our main result concerning the computation of the atoms generated by a collection of sets.

Theorem 1. Given a space set H and a collection $\mathcal{R}$ of n subsets of H, the collection $\mathcal{UC}(\mathcal{R})$ of combinations that canonically represent the atoms $\mathcal{A}(\mathcal{R})$ can be incrementally computed with $O(\min(n + k\overline{K} \log m, nm)m)$ elementary set operations where: m is the number of atoms generated by $\mathcal{R}$; k is the overlapping degree of $\mathcal{R}$; $\overline{K}$ is the average overlapping degree of $\mathcal{C}(\mathcal{R})$; $\overline{k}$ is the average overlapping degree of $\mathcal{R}$. Within this computation, each combination $c \in \mathcal{UC}(\mathcal{R})$ can be associated to the list $\mathcal{R}(c)$ of sets in $\mathcal{R}$ that contain c. If sets are represented by ℓ-wildcard expressions (resp. (d, ℓ)-multi-ranges), the representation can be computed in $O(\ell \min(n + k\overline{K}, nm)m)$ (resp. $O(\ell \min(\overline{k} \log^d m + k\overline{K}, nm)m))$ time.

Application to forwarding loop detection Theorem 1 has the following consequences for forwarding loop detection.

Corollary 1. Given a network $\mathcal{N}$ with collection $\mathcal{R}$ of n rule sets with T_ℓ-bounded representation, forwarding loop detection can be performed in $O(T_\ell \min(n + k\overline{K} \log m, nm)m + \overline{k} n_G m)$ time where m is the number of atoms in $\mathcal{A}(\mathcal{R})$, k is the overlapping degree of $\mathcal{R}$ and $\overline{k}$ (resp. $\overline{K}$) is the average overlapping degree of $\mathcal{R}$ (resp. $\mathcal{C}(\mathcal{R})$). If sets are represented by ℓ-wildcard expressions (resp. (d, ℓ)-multi-ranges), the representation can be computed in $O(\ell \min(n + k\overline{K}, nm)m + \overline{k} n_G m)$ (resp. $O(\ell \min(\overline{k} \log^d m + k\overline{K}, nm)m + \overline{k} n_G m))$ time.

This result can be extended to handle write actions as explained in Appendix F due to the lack of space. The upper-bounds for forwarding loop detection listed in Table 1 follow from Corollary 1 which is proved in Appendix A.3. A key ingredient consists in maintaining for each rule set a list that describes its presence, priority and action for each node. Detecting a loop for a header class a then consists in merging the lists associated to the rule sets containing a (as provided by our atom representation) for obtaining the forwarding graph $G_a = G_h$ for all $h \in a$. Directed cycle detection is finally performed on each such graph.

In the rest of this Section, we prove Theorem 1 by introducing two incremental algorithms for atom computation and analyzing their performance. The incremental approach allows to avoid exponential blow-up even in cases where the number of combinations can be exponential in the number m of atoms (an example is given in Appendix B.2).

Algorithms for updating atoms We first propose a basic algorithm for updating the collection $\mathcal{UC}$ of uncovered combinations of a collection $\mathcal{R}$ when a set r is added to $\mathcal{R}$. The main idea is that after adding r to $\mathcal{R}$, the only new uncovered combinations that can be created are intersections of pre-existing uncovered combinations with r (see Lemma 1 in Appendix A.2). We thus first add to $\mathcal{UC}$ the combinations $c \cap r$ for $c \in \mathcal{UC}$. As this may introduce covered combinations, we then compute the atom size $c.atsize = |a(c)|$ for each combination c. It is then sufficient to remove any combination c with $c.atsize = 0$ to finally obtain $\mathcal{UC}(\mathcal{R} \cup \{r\})$. This atom size computation is possible because we have $d = \cup_{c \in \mathcal{UC} | c \subseteq d} a(c)$ for all $d \in \mathcal{UC}$. (see Lemma 2 in Appendix A.2). As this union is disjoint, we have $|a(d)| = |d| - \sum_{c \in \mathcal{UC} | c \subsetneq d} |a(c)|$. We thus compute the inclusion relation between combinations and store in $c.sup$ the combinations that strictly contain c. Initializing $c.atsize$ to $|c|$ for all c, we then scan all combinations c by non-decreasing cardinality (or any topological order for inclusion) and subtract $c.atsize$ from $d.atsize$

for each $d \in c.sup$. A simple proof by induction allows to prove that $c.atsize = |a(c)|$ when c is scanned. The whole process in summarized in Algorithm 1.

```
Procedure $basicAdd(r, \mathcal{R}, \mathcal{UC})$
    $\mathcal{UC}' := \mathcal{UC} \cup \{c \cap r \mid c \in \mathcal{UC}\}$
    For each $c \in \mathcal{UC}'$ do
        $c.atsize := |c|$
        $c.sup := \{d \in \mathcal{UC}' \mid c \subsetneq d\}$
    For each $c \in \mathcal{UC}'$ in non-decreasing cardinality order do
        For each $d \in c.sup$ do  $d.atsize := d.atsize - c.atsize$
    $\mathcal{UC} := \mathcal{UC}' \setminus \{c \in \mathcal{UC}' \mid c.atsize = 0\}$
```

Algorithm 1: Add a set r to a collection $\mathcal{R}$ and update the collection $\mathcal{UC} = \mathcal{UC}(\mathcal{R})$ of its uncovered combinations accordingly.

The correctness of Algorithm 1 follows from the two above remarks (that is Lemma 1 and Lemma 2 in Appendix A.2). Its main complexity cost comes from intersecting r with each combination and computing the inclusion relation between combinations, that is $O(nm)$ and $O(m^2)$ elementary set operations respectively. Starting from $\mathcal{UC} = \{H\}$ and incrementally applying Algorithm 1 to each set in $\mathcal{R}$ thus allows to obtain $\mathcal{UC}(\mathcal{R})$ with $O(nm^2)$ elementary set operations, yielding the general bound of Theorem 1.

To derive better bounds for low overlapping degree k, we propose a more involved algorithm that maintains $c.sup$ and $c.atsize$ from one iteration to another and makes only the necessary updates. This requires to handle several subtleties to enable lower complexity.

We similarly start by computing the collection $Inter = \{c \in \mathcal{UC} \mid c \cap r \neq \emptyset\}$ of combinations intersecting r. A first subtlety comes from the fact that several combinations c may result in the same $c' = c \cap r$. However, we are only interested in the combination c which is minimal for inclusion that we call the *parent* of c'. The reason is that $c'.sup$ can then be computed from $c.sup$. The parent is unique unless c' is covered in which case c' is marked as covered and discarded (see the argument for $c'.sup$ computation later on). To obtain right parent information, we thus process all $c \in Inter$ by non-decreasing cardinality. The produced combinations $c' = c \cap r$ such c' were not in $\mathcal{UC}$ are called *new* combinations. Their atom size is initialized to $c'.atsize = |c'|$. See the "Parent computation" part of Algorithm 2.

We then remark that we only need to compute (or update) $c.sup$ for combinations that include r, which we store in a set *Incl*. We also note that $c.atsize$ needs to be computed when c is new and updated when c is the parent of a new combination. A second subtlety resides in computing (or updating) $c.sup$ only when c is not covered that is when $c.atsize$ (after computation) appears to be non-zero. As the computation of $c.sup$ lists is the most heavy part of the computation, this is necessary to enable our complexity analysis. For that purpose, we scan *Incl* by non-decreasing cardinality so that the correct value of $c.atsize$ is known when c is scanned similarly as in Algorithm 1. However, we avoid any useless computation when $c.atsize$ is zero. Otherwise, we compute (or update) $c.sup$ and decrease $d.atsize$ by $c.atsize$ from $d \in c.sup$ for adequate d: if c and d where both in $\mathcal{UC}$, this computation has already been made; it is only necessary when d is new or when d is the parent of c. We optionally maintain for each combination c a list $c.cont$ that contain the list of sets $r \in \mathcal{R}$ that contain c (Such lists are not necessary for the computation but they are useful for loop detection as detailed in Appendix A.3). See the "Atom size computation" part of Algorithm 2.

A last critical point resides in the computation of $c'.sup$ for each new combination c'. The $c'.sup$ list can be obtained from $c'.parent.sup$ by copying and also intersecting elements of $c'.parent.sup$ with r. This is sufficient: for $d \in \mathcal{UC}$ such that $c' \subsetneq d \cap r$, we can consider $c = c'.parent \cap d$. If $c \in \mathcal{UC}$, $c'.parent = c$ by minimality of $c'.parent$ and we thus have $d \in c'.parent.sup$. The case where $c \notin \mathcal{UC}$ and $d \notin c'.parent.sup$ cannot happen as it would imply that two different combinations $c_1 = c'.parent$ and $c_2 \subseteq c$ generate c' by intersection with r ($c_1 \cap r = c_2 \cap r = c'$) and are both minimal for inclusion. In such case, $c_1 \cap c_2$ was covered in $\mathcal{R}$ and so would be c' in

```
Procedure $add(r, \mathcal{R}, \mathcal{UC} = \mathcal{UC}(\mathcal{R}))$
    $New := \emptyset$; $Incl := \emptyset$;
    /* ---------------------- Parent computation ----------------------          */
    $Inter := \{c \in \mathcal{UC} \mid c \cap r \neq \emptyset\}$
    Sort $Inter$ by non-decreasing cardinality.
    For each $c \in Inter$ do
        $c' := c \cap r$
        If $c' \notin Incl$ then
            If $c' \notin \mathcal{UC}$ then
                $\mathcal{UC} := \mathcal{UC} \cup \{c'\}$; $New := New \cup \{c'\}$;
                $c'.atsize := |c'|$; $c'.sup := \{\}$; $c'.cont := \{\}$  /* Updated later.          */
            $c'.parent := c$; $c'.covered := false$; $Incl := Incl \cup \{c'\}$;
        else
            If $c'.parent \not\subseteq c$ then $c'.covered := true$
    Remove from $Incl$, $New$ and $\mathcal{UC}$ all $c$ such that $c.covered = true$.
    /* ---------------------- Atom size computation ----------------------       */
    Sort $Incl$ by non-decreasing cardinality.
    For each $c \in Incl$ do
        If $c.atsize > 0$ then
            /* Adjust $c.sup$, $c.cont$ and update $d.atsize$ for impacted $d \supsetneq c$:          */
            If $c \in New$ then
                $c.parent.atsize := c.parent.atsize - c.atsize$
                $c.sup := \{c.parent\} \cup c.parent.sup$
                $c.cont := c.parent.cont$
            $c.sup := c.sup \cup \{d \cap r \mid d \in c.sup$ and $d \cap r \in Incl \setminus \{c\}\}$
            $c.cont := c.cont \cup \{r\}$
            For each $d \in c.sup$ s.t. $d \in New$ do
                $d.atsize := d.atsize - c.atsize$
    /* ---------------------- Remove covered combinations ----------------------
       */
    For each $c \in Incl$ do
        Remove from $c.sup$ any $d$ such that $d.atsize = 0$.
        If $c.atsize = 0$ then $\mathcal{UC} := \mathcal{UC} \setminus \{c\}$; $Incl := Incl \setminus \{c\}$;
        If $c \in New$ and $c.parent.atsize = 0$ then $\mathcal{UC} := \mathcal{UC} \setminus \{c.parent\}$
```

Algorithm 2: Add a set r to a collection $\mathcal{R}$ and update the collection $\mathcal{UC}$ of its uncovered combinations accordingly.

$\mathcal{R}$ (and also in $\mathcal{R} \cup \{r\}$). That is why such combination c' are already discarded during parent computation. On the other hand, the list $c.sup$ of a combination $c \in \mathcal{UC}$ can be updated by intersecting elements of $c.sup$ with r: when $c \subsetneq d \cap r$ for $c \subseteq r$, we have $c \subsetneq d$.

Finally, combinations c with $c.atsize = 0$ are discarded and removed from $b.sup$ list of remaining combinations as detailed in the "Remove covered combinations" part of Algorithm 2.

The main argument in the complexity analyzis of Algorithm 2 comes from bounding the size of each $c.sup$ list by $|\mathcal{UC}(a(c))|$ (this is where the overlapping degree of $\mathcal{C}(\mathcal{R})$ is used). The number of elementary set operations performed is indeed $O(m + \sum_{a \in A_r} |\mathcal{UC}'(a)| \log m)$ where m denotes the number of atoms of $\mathcal{R}$, $A_r = \{a \in \mathcal{A}(\mathcal{R}') \mid a \subseteq r\}$ denotes the atoms of $\mathcal{R}' = \mathcal{R} \cup \{r\}$ included in r, and $\mathcal{UC}'(a)$ denotes the uncovered combinations of $\mathcal{R}'$ that contain a. A key step consists in proving $|Inter| \leq \sum_{a \in A_r} |\mathcal{UC}'(a)|$. The $\log m$ accounts for membership tests and add/remove operations on collections of combinations. In the case of ℓ-wildcard and (d, ℓ)-multi-ranges, this factor can be saved by storing collections of sets in tries rather than balanced binary search trees. With (d, ℓ)-multi-ranges, a segment tree allows to retrieve $Inter$ in $O(|Inter| + \log^d m)$ elementary set operations [2, 8].

The various bounds for low overlapping degree of Theorem 1 results from applying iteratively Algorithm 2 for each set of $\mathcal{R}$ and by carefully bounding the sums of $\left|\mathcal{UC}'(a)\right|$ terms. Intuitively, each atom a generated by the collection $\mathcal{R}_i$ of the first i sets of $\mathcal{R}$ can be associated to an atom a' of $\mathcal{R}$ such that $\left|\mathcal{UC}'(a)\right| \leq |\mathcal{C}(a')|$ where $\mathcal{C} = \mathcal{C}(\mathcal{R})$. As each atom is included in k sets of $\mathcal{R}$ at most, this allows to bound the overall sum of $\left|\mathcal{UC}'(a)\right|$ terms by $k\overline{K}m$. Details are given in Appendix A.2.

5 Comparison with previous works

We give in Appendix C more details about "linear fragmentation", a phenomenon observed by Kazemian et al. [14], and propose small overlap degree as a plausible cause. The notion of uncovered combination is linked to that of weak completion introduced by [5] in the context of rule-conflict resolution as detailed in Appendix D. We now provide examples where use of set complement computation can lead to exponential blow-up in previous work.

5.1 Veriflow

Veriflow [17] incrementally computes a partition into sub-classes that forms a refinement of the header classes: when a rule r is added, each sub-class c is replaced by $c \cap r$ and a partition of $c \setminus r$. Veriflow benefits from the hypothesis that headers can be decomposed into d fixed fields and that each rule set can be represented by a multi-range $r = [a_1, b_1] \times \cdots \times [a_d, b_d]$. The intersection of two multi-ranges is obviously a multi-range. However, set difference is obtained by intersection with the complement which is represented as the union of up to $2d - 1$ multi-ranges. We prove in Appendix B.1 that successive difference with rules of the form $[O, \infty]^{i-1} \times [a_j, a_j] \times [b, b]^{d-i}$ with $0 < a_1, \ldots, a_p < b$ can generate $\Omega(\left(\frac{n}{d}\right)^d \frac{m}{d})$ multi-ranges in the computations of Veriflow as indicated in Table 1.

5.2 HSA / NetPlumber

HSA/NetPlumber [15, 13] use clever heuristics to efficiently compute the set of headers H_P than can traverse a given path P. An important one consists in lazy subtraction: set difference computations are postponed until the end of the path. For that purpose, this set H_P is represented as a union of terms of the form $s = c_0 \setminus \cup_{i=1..p} c_i$ where the elementary sets $c_0, \ldots, c_p$ are represented with wildcards. The emptiness of such terms is regularly tested. A simple heuristic is used during the construction of the path: s is obviously empty if c_0 is included in c_i for some $i \geq 1$. But if the path loops, HSA has to develop the corresponding terms into a union of wildcards to determine if one of them may produce a forwarding loop.

We now provide an example where this emptiness test can take exponential time. Consider a node whose forwarding table consists in $\ell + 1$ rules with following rule sets:

$$r_0 = 1^\ell, \quad r_i = 1^{\ell-i} 0 *^{i-1} \text{ for } i = 1..\ell, \quad \text{and} \quad r_{\ell+1} = *^\ell.$$

All rules are associated with the drop action except the last rule (with rule set $r_{\ell+1}$) whose action is to forward to the node itself. Such a forwarding table is depicted in Figure 1 for $\ell = 4$. Starting a loop detection from that node, HSA detects a loop for headers in $r_{\ell+1} \setminus \cup_{i=0..\ell} r_i$. The emptiness of this term is thus tested. For that purpose, HSA represents the complement of r_i with $0 *^{\ell-1} \cup * 0 *^{\ell-2} \cup \cdots \cup *^{\ell-i-1} 0 *^i \cup *^{\ell-i} 1 *^{i-1}$. Note that each of the $\ell - i$ wildcard expressions in that union have only one non-$*$ letter. Distributivity is then used to compute $r_{\ell+1} \setminus \cup_{i=0..\ell} r_i$ as $\overline{r_0} \cap \cdots \cap \overline{r_\ell}$. After expanding the first $j - 1$ intersections, HSA thus obtains a union of wildcards with j letters in $\{0, 1\}$ and $\ell - j$ letters equal to $*$ that has to be intersected with $\overline{r_{j+1}} \cap \cdots \cap \overline{r_\ell}$. In particular, this unions contains all strings with j letters equal to 0 and $\ell - j$ equal to $*$. All ℓ-letter strings with alphabet $\{*, 0\}$ are produced during the computation which thus requires $\Omega(\ell 2^\ell)$ time. For testing a network with n_G similar nodes, HSA thus requires time $\Omega(\ell n_G 2^\ell)$. As all sets $r_0, \ldots, r_\ell$ are pairwise disjoint, the overlapping degree of the collection is $k = 2$ and this justifies the two lower-bounds indicated for NetPlumber in Table 1.

6 Future work

Our approach could be naturally integrated in the Veriflow [17] framework both for speed-up and performance-guarantee considerations. Our ideas can also be integrated in the NetPlumber [13] framework (see the similar approach proposed in Appendix E). This would allow to enhance the emptiness tests performed within NetPlumber to guarantee polynomial time execution when the number of header classes is polynomially bounded. In the context of multi-ranges, the emptiness test of expressions is equivalent to the Klee measure problem which consists in computing the volume of a union of boxes. Indeed, expression $r_p \setminus \cup_{i=1..p-1} r_i$ is empty when the volume of the union of the boxes $r_1 \cap r_p, \ldots, r_{p-1} \cap r_p$ equals that of r_p. According to recent work [6], the complexity of this problem is believed to be $\Theta(n^{d/2})$. It would be interesting to determine if such low complexity bounds extend to atom computation in the case of multi-ranges rules.

References

[1] Hari Adiseshu, Subhash Suri, and Guru M. Parulkar. Detecting and resolving packet filter conflicts. In *Proceedings IEEE INFOCOM 2000, The Conference on Computer Communications, Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies, Reaching the Promised Land of Communications, Tel Aviv, Israel, March 26-30, 2000*, pages 1203–1212. IEEE, 2000.

[2] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational Geometry: Algorithms and Applications.* Springer-Verlag TELOS, Santa Clara, CA, USA, 3rd ed. edition, 2008.

[3] Allan Borodin, Rafail Ostrovsky, and Yuval Rabani. Lower bounds for high dimensional nearest neighbor search and related problems. In Jeffrey Scott Vitter, Lawrence L. Larmore, and Frank Thomson Leighton, editors, *Proceedings of the Thirty-First Annual ACM Symposium on Theory of Computing, May 1-4, 1999, Atlanta, Georgia, USA*, pages 312–321. ACM, 1999.

[4] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 99–110, New York, NY, USA, 2013. ACM.

[5] Matthieu Boutier and Juliusz Chroboczek. Source-specific routing. In *IFIP Networking*, 2015.

[6] Timothy M. Chan. Klee's measure problem made easy. In *54th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2013, 26-29 October, 2013, Berkeley, CA, USA*, pages 410–419. IEEE Computer Society, 2013.

[7] Moses Charikar, Piotr Indyk, and Rina Panigrahy. New algorithms for subset query, partial match, orthogonal range searching, and related problems. In Peter Widmayer, Francisco Triguero Ruiz, Rafael Morales Bueno, Matthew Hennessy, Stephan Eidenbenz, and Ricardo Conejo, editors, *Automata, Languages and Programming, 29th International Colloquium, ICALP 2002, Malaga, Spain, July 8-13, 2002, Proceedings*, volume 2380 of *Lecture Notes in Computer Science*, pages 451–462. Springer, 2002.

[8] Herbert Edelsbrunner and Hermann A. Maurer. On the intersection of orthogonal objects. *Information Processing Letters*, 13(4):177–181, 1981.

[9] David Eppstein and S. Muthukrishnan. Internet packet filter management and rectangle geometry. In S. Rao Kosaraju, editor, *Proceedings of the Twelfth Annual Symposium on Discrete Algorithms, January 7-9, 2001, Washington, DC, USA.*, pages 827–835. ACM/SIAM, 2001.

[10] Anja Feldmann and S. Muthukrishnan. Tradeoffs for packet classification. In *Proceedings IEEE INFOCOM 2000, The Conference on Computer Communications, Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies, Reaching the Promised Land of Communications, Tel Aviv, Israel, March 26-30, 2000*, pages 1193–1202. IEEE, 2000.

[11] Pankaj Gupta and Nick McKeown. Algorithms for packet classification. *IEEE Network: The Magazine of Global Internetworking*, 15(2):24–32, March 2001.

[12] Naga Praveen Katta, Jennifer Rexford, and David Walker. Incremental consistent updates. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN '13, pages 49–54, New York, NY, USA, 2013. ACM.

[13] Peyman Kazemian, Michael Chang, Hongyi Zeng, George Varghese, Nick Mckeown, Scott Whyte, and U C San Diego. Real Time Network Policy Checking using Header Space Analysis. In *NSDI*, 2013.

[14] Peyman Kazemian, George Varghese, and Nick McKeown. Header space analysis: Static checking for networks. Technical report, Stanford, 2011. http://yuba.stanford.edu/ peyman/docs/headerspace_tech_report.pdf.

[15] Peyman Kazemian, George Varghese, and Nick McKeown. Header space analysis: Static checking for networks. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 113–126, San Jose, CA, 2012. USENIX.

[16] Kazemian Peyman. HSA/NetPlumber source code repository. `https://bitbucket.org/peymank/hassel-public/`.

[17] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P Brighten Godfrey. VeriFlow : Verifying Network-Wide Invariants in Real Time. In *NSDI*, 2013.

[18] Vasileios Kotronis, Xenofontas Dimitropoulos, and Bernhard Ager. Outsourcing the routing control logic: Better internet routing based on SDN principles. In *Proceedings of the 11th ACM Workshop on Hot Topics in Networks*, HotNets-XI, pages 55–60, New York, NY, USA, 2012. ACM.

[19] Haohui Mai, Ahmed Khurshid, P Brighten Godfrey, and Samuel T King. Debugging the Data Plane with Anteater. In *SIGCOMM*, 2011.

[20] Arnaud Mary and Yann Strozecki. Efficient enumeration of solutions produced by closure operations. In Nicolas Ollinger and Heribert Vollmer, editors, *33rd Symposium on Theoretical Aspects of Computer Science, STACS 2016, February 17-20, 2016, Orléans, France*, volume 47 of *LIPIcs*, pages 52:1–52:13. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016.

[21] Christopher Monsanto, Joshua Reich, Nate Foster, Jennifer Rexford, and David Walker. Composing software-defined networks. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, nsdi'13, pages 1–14, Berkeley, CA, USA, 2013. USENIX Association.

[22] Mihai Patrascu. Unifying the landscape of cell-probe lower bounds. *SIAM J. Comput.*, 40(3):827–847, 2011.

[23] Peter Perešíni, Maciej Kuzniar, Nedeljko Vasić, Marco Canini, and Dejan Kostic. OF.CPP: Consistent Packet Processing for Openflow. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN '13, pages 97–102, New York, NY, USA, 2013. ACM.

[24] Ronald L. Rivest. Partial-match retrieval algorithms. *SIAM J. Comput.*, 5(1):19–50, 1976.

[25] Route Views Project. BGP traces. `http://routeviews.org/`.

[26] Matteo Varvello, Rafael Laufer, Feixiong Zhang, and T.V. Lakshman. Multi-layer packet classification with graphics processing units. In *CoNEXT*, 2014.

[27] Hongyi Zeng, Shidong Zhang, Fei Ye, Vimalkumar Jeyakumar, Mickey Ju, Junda Liu, Nick McKeown, and Amin Vahdat. Libra: Divide and conquer to verify forwarding tables in huge networks. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, pages 87–99, Berkeley, CA, USA, 2014. USENIX Association.

Appendix A : Proof details of Theorem 1 and Corollary 1

We present here a refined version of Theorem 1 with more hypothesis about the data-structure used to store a collection of sets. We first review these hypothesis and then the proof details of both versions.

A.1 Collection of sets representation

When manipulating a collection of p sets in $\mathcal{D}$, we assume that their representations are stored in a collection data-structure allowing to dynamically add, remove or test membership of a set. Such operation are called *p-collection operations*. We can use a balanced binary search tree when comparisons according to a total order can be performed. Such comparison can usually be obtained by comparing directly the binary representations themselves of the set in linear time (and thus $O(T_H)$ for sets with T_H-bounded representation). It is considered as an elementary set operation. In the case of wildcard expressions, the complexity of these operations can be reduced to $O(\ell)$ time by using a trie or a Patricia tree. Our algorithms will also make use of an operation similar to stabbing query that we call *p-intersection* query. It consists in producing the list L_s of sets in a collection $\mathcal{R}$ of p sets that intersect a given query set s ($L_s = \{r \in \mathcal{R} \mid s \cap r \neq \emptyset\}$). We additionally require that the list L_s is topologically sorted according to inclusion. We say that p-intersection queries can be answered with *overhead* $(T_{inter}(p), T_{updt}(p))$ when dynamically adding or removing a set from the collection takes time $T_{updt}(p)$ at most and the p-intersection query for any set $s \in \mathcal{D}$ takes time $T_{inter}(p) + |L_s|T_H$ at most. In the case of d-dimensional multi-ranges, a segment tree allows to answer p-intersection queries with overhead $(O(\log^d p), O(\log^d p))$ [2, 8]. In the case of wildcard expressions, a trie or a Patricia tree allows to answer p-intersection queries with overhead $(O(\ell p), O(\ell))$ (the whole tree has to be traversed in the worse case, but no sorting is necessary as the result is naturally obtained according to lexicographic order).

A.2 Incremental computation of atoms (Theorem 1)

The following is a refinement of Theorem 1 with respect to data-structures that provide time bounds on p-collection operations and p-intersection queries. For the sake of simplicity of asymptotic expressions, we make the very loose assumption that $\ell = o(m)$ and $n \leq m$. (We are mainly interested in the case where m is large. Note also that examples with $m < n$ would be very peculiar.)

Theorem 2. Given a space set H and a collection $\mathcal{R}$ of n subsets of H, the collection $\mathcal{UC}(\mathcal{R})$ of combinations that canonically represent the atoms generated by $\mathcal{R}$ can be incrementally computed with $O(\min(n+k\overline{K}\log m, \overline{k}m\log m, nm)m)$ elementary set operations where m denotes the number of atoms generated by $\mathcal{R}$, k denotes the overlapping degree of $\mathcal{R}$, $\overline{k}$ denotes the average overlapping degree of $\mathcal{R}$ and $\overline{K}$ denotes the average overlapping degree of $\mathcal{C}(\mathcal{R})$.

More precisely, if the data-structures used for representing sets and collections of sets enable elementary set operations within time T_{set}, p-collection operations within time $T_{coll}(p)$ and p-intersection queries with overhead $(T_{inter}(p), T_{updt}(m))$, then the representation of the atoms generated by $\mathcal{R}$ can be computed in $O(nT_{inter}(m) + \overline{k}mT_{updt}(m) + \min(k\overline{K}, km)m(T_{set} + T_{coll}(m)))$ time.

The following lemma formally states that uncovered combinations of $\mathcal{R} \cup \{r\}$ can be obtained from $\mathcal{UC}(\mathcal{R})$ and justifies the overall approach of Algorithm 2.

Lemma 1. Given a new rule $r \subseteq H$, the collection $\mathcal{UC}(\mathcal{R}')$ of uncovered combinations of $\mathcal{R}' = \mathcal{R} \cup \{r\}$ can be obtained by intersecting uncovered combinations in $\mathcal{UC}(\mathcal{R})$ with r. More precisely, we have $\mathcal{UC}(\mathcal{R}') \subseteq \mathcal{UC}(\mathcal{R}) \cup \{c \cap r | c \in \mathcal{UC}(\mathcal{R})\}$.

Proof. Consider an uncovered combination $c' \in \mathcal{UC}(\mathcal{R}')$. Let c be the intersection of the containers of c' in $\mathcal{R}$: $c = \cap_{r \in \mathcal{R}(c')} r$. We have either $c' = c$ if $\mathcal{R}'(c') = \mathcal{R}(c')$ or $c' = c \cap r$ if $\mathcal{R}'(c') = \mathcal{R}(c') \cup r$. To conclude, it is thus sufficient to show that c is uncovered in $\mathcal{R}$. This follows from $c' \subseteq c$ and

$\mathcal{R}'(c') \subseteq \mathcal{R}(c) \cup \{r\}$: the non-containers of c in $\mathcal{R}$ are also non-containers of c' in $\mathcal{R}'$ and if c was covered, so would be c'. □

The following Lemma, which expresses any combination as a disjoint union of atoms justify the computation of atom cardinalities in Algorithm 2 by scanning *c.sup* lists for $c \in \mathcal{UC}$.

Lemma 2. Given a combination collection $\mathcal{C}' \subseteq \mathcal{C}(\mathcal{R})$ containing $\mathcal{UC}(\mathcal{R})$, we have $d = \cup_{c\in\mathcal{C}'|c\subseteq d} a(c)$ for all $d \in \mathcal{C}'$. This union is disjoint and we have $|a(d)| = |d| - \sum_{c\in\mathcal{C}'|c\subsetneq d} |a(c)|$.

Proof. Reminding that any combination d includes $a(d) = d \cap \big(\cap_{r\in\mathcal{R}\setminus\mathcal{R}(d)} \overline{r}\big)$, we have $\cup_{c\in\mathcal{C}'|c\subseteq d} a(c) \subset d$. Conversely, consider $h \in d$. The sets of $\mathcal{R}$ containing h are $\mathcal{R}(\{h\})$, and we have $h \in a(c)$ for $c = \cap_{r\in\mathcal{R}(\{h\})} r$. As $\mathcal{R}(d) \subseteq \mathcal{R}(\{h\})$ (the sets that contain d also contain h) and $d = \cap_{r\in\mathcal{R}(d)} r$, we have $c \subseteq d$. Hence $d \subset \cup_{c\in\mathcal{C}'|c\subseteq d} a(c)$. This union is disjoint as each $a(c)$ is either an atom or is empty. □

We can now state the following proposition about the guarantees of Algorithm 2.

Proposition 2. Algorithm 2 allow to dynamically update the collection $\mathcal{UC}$ of uncovered combinations of a collection $\mathcal{R}$ using $O(m + \sum_{a\in A_r} \left|\mathcal{UC}'(a)\right| \log m)$ elementary set operations when a rule r is added to $\mathcal{R}$, where m denotes the number of atoms of $\mathcal{R}$, $A_r = \{a \in \mathcal{A}(\mathcal{R}') \mid a \subseteq r\}$ denotes the atoms of $\mathcal{R}'$ included in r, and $\mathcal{UC}'(a)$ denotes the uncovered combinations of $\mathcal{R}'$ that contain a.

More precisely, if the data-structures used for representing sets and collections of sets enable elementary set operations within time T_{set}, p-collection operations within time $T_{coll}(p)$ and p-intersection queries with overhead $(T_{inter}(p), T_{updt}(m))$, then the update of $\mathcal{UC}$ can be performed in $O(T_{inter}(m) + |A_r|\, T_{updt}(m) + (T_{set} + T_{coll}(m)) \sum_{a\in A_r} \left|\mathcal{UC}'(a)\right|)$ time.

Proof. As discussed before the correctness of Algorithm 2 for obtaining uncovered combinations after adding set r to a collection $\mathcal{R}$ from $\mathcal{UC} = \mathcal{UC}(\mathcal{R})$ and $\{c \cap r \mid c \in \mathcal{UC}\}$ results from Lemma 1. For a new combination c, the *c.sup* list is obtained from *c.parent.sup* and *c.sup* is updated similarly for $c \in \mathcal{UC}$ such that $c \subseteq r$. The correctness of this approach has already been discussed in Subsection 4. We develop here the key argument for ignoring a combination $c' = c \cap r$ when it is produced by several minimal elements $c_1, \ldots, c_i \in \mathcal{UC}$ such that $c_j \cap r = c'$ for j in $1..i$. If this happens, we know that $\cap_{j\in 1..i} c_j$ is not in $\mathcal{UC}$, meaning that it is covered in $\mathcal{R}$ and so is $c' \subseteq \cap_{j\in 1..i} c_j$ in $\mathcal{R} \cup \{r\}$. We can thus safely eliminate c' in the first phase of the algorithm. For the remaining new combinations c', the parent c of c' is the unique combination $c \in \mathcal{UC}$ such that $c \cap r = c'$ and which is minimal for inclusion.

The correctness of the atom cardinality computation follows by induction on the number combinations in *Incl* processed so far in the corresponding for loop. Consider a newly created combination c. The initial value of *c.atsize* is $|c|$. Assuming that the correct value *b.atsize* has been obtained for b processed before c and $c \in$ *b.sup*, $|a(b)|$ has been subtracted from *c.atsize* and Lemma 2 implies that *c.atsize* $= |a(c)|$ when we consider c in the for loop. For c already in $\mathcal{UC}$ before adding r and for $b \subseteq c$ processed before c, *b.atsize* has been subtracted from *c.atsize* only for newly created b. For $b \in \mathcal{UC}$, $|a(b)|$ may have decreased but this difference is compensated by $\sum_{b'\in New|b'\subsetneq b} |a(b')|$. This is the reason why Algoritihm 2 updates only *c.parent.atsize* besides $d \in$ *c.sup* such that $d \in$ *New*. The correctness of the atom cardinality computation implies that all covered combinations are removed and the correctness of Algorithm 2 follows.

We now analyze the complexity of Algorithm 2. The bound in terms of elementary set operations is obtained when balanced binary search tree (BST for short) are used to store the various collections of sets (i.e. $\mathcal{UC}$, *Incl*, *New* and *c.sup* for $c \in \mathcal{UC}$). When adding a set r, finding the combinations in $\mathcal{UC}$ that intersect r is a m-intersection query and can be performed in $O(T_{inter}(m) + |Inter| T_{set})$ time or $O(m \log m)$ set operations using BST (sorting is only necessary in that case). The collection *Incl* is then constructed in $O(|Inter| \log m)$ operations with BST or $O(|Inter|(T_{set} + T_{coll}(m)))$ with appropriate data-structures. Removing combinations c such that *c.covered* $=$ *true* is just a matter of scanning *Incl* again and can be done within the same complexity. Let $\mathcal{I}$ denote the combinations included in *Incl* at that point (just before cardinality computations). The computation of *c.sup* for $c \in \mathcal{I}$ is done only when $c.atsize > 0$, i.e. only if c represents one of the atoms in A_r. we thus have $|I| \leq |A_r|$. This requires at most $O(|c.parent.sup|)$ operations. Note that for each uncovered combination $d \in c.parent.sup$ yields at least one uncovered combination in *c.sup* (d itself or $d \cap r$ or both). We thus have $|c.parent.sup| \leq \left|\mathcal{UC}'(a(c))\right|$. The overall computation of *sup* lists can thus be performed within $O(\sum_{a \in A_r} \left|\mathcal{UC}'(a)\right| \log m)$ set operations with BST and $O((T_{set} + T_{coll}(m)) \sum_{a \in A_r} \left|\mathcal{UC}'(a)\right|)$ time with appropriate data-structures. The computation of class cardinalities and the removal of covered combinations from the *sup* lists have same complexity. Removal of covered combinations from $\mathcal{UC}$ and *Incl* takes $O(|A_r|)$ collection operations and fits within the same complexity bound. Additional cost of $|A_r|\, T_{updt}(m)$ is necessary when maintaining data-structures enabling efficient m-intersection queries. The whole algorithm can thus be performed in $O(T_{inter}(m) + |A_r|\, T_{updt}(m) + |Inter|(T_{set} + T_{coll}(m)) + (T_{set} + T_{coll}(m)) \sum_{a \in A_r} \left|\mathcal{UC}'(a)\right|)$ time or using $O(m + |Inter| \log m + \sum_{a \in A_r} \left|\mathcal{UC}'(a)\right| \log m)$ elementary set operations with BST.

To achieve the proof of complexity of Algorithm 2, we show $|Inter| \leq \sum_{a \in A_r} \left|\mathcal{UC}'(a)\right|$. Consider a combination $c \in \mathcal{UC}$ that intersects r. Then c can be associated to an atom *c.atm* of A_r included in $c \cap r$ (such atoms exist according to Lemma 2). For any atom $a \in A_r$, let *a.par* denote the atom in $\mathcal{A}(\mathcal{R})$ that contains a (we have $a = a.par$ or $a = a.par \cap r$). Now for $c \in Inter$, consider the atom $a = c.atm \in A_r$. As $a.par \in \mathcal{A}(\mathcal{R})$ is an atom and $c \in \mathcal{C}(\mathcal{R})$ is a combination intersecting *a.par*, we have $a.par \subseteq c$ and $c \in \mathcal{UC}(a.par)$ is one of the combinations in $\mathcal{UC}(\mathcal{R})$ that contains *a.par*. For each such combination c, $a(c)$ intersects r or $\overline{r}$ (both), and c or $c \cap r$ is uncovered in $\mathcal{R}' = \mathcal{R} \cup \{r\}$. Both contain a and we have $|\mathcal{UC}(a.par)| \leq \left|\mathcal{UC}'(a)\right|$. We can thus write $|Inter| = \sum_{a \in A_r} |\{c \in Inter \mid c.atm = a\}| \leq \sum_{a \in A_r} |\mathcal{UC}(a.par)| \leq \sum_{a \in A_r} \left|\mathcal{UC}'(a)\right|$. □

It should be noted that in the case of (d, ℓ)-multi-ranges, we can distinguish in the analysis, the computation of $|c|$ for new c that costs $O(\ell \log \ell)$ from other cardinality manipulations (subtractions and comparisons) that take $O(\ell)$. This allows to obtain the bound claimed in Theorem 1.

Theorem 2 results from iteratively applying Algorithm 2 (or Algorithm 1) for each set of $\mathcal{R} = \{r_1, \ldots, r_n\}$.

Proof of Theorem 2. From Proposition 2, the overall complexity of atom computation is $O(\sum_{i=1..n} \left(m_i + \sum_{a \in A_i} |\mathcal{UC}_i(a)|\right) \log m)$ set operations where m_i denotes the number of atoms in $\mathcal{A}(\{r_1, \ldots, r_{i-1}\})$ and A_i denotes the atoms of $\mathcal{A}(\{r_1, \ldots, r_i\})$ included in r_i and $\mathcal{UC}_i = \mathcal{UC}(\{r_1, \ldots, r_i\})$. We first consider the case where $\overline{K}$ is unbounded (it is possible to construct examples with $m = n$ and $\overline{K} = \Omega(2^n)$). As we add a set to $\mathcal{R}$, the number of atoms can only increase (each atom remains unchanged or is eventually split into two). We thus have $m_i \leq m$ and $|A_i| \leq |\{a \in \mathcal{A}(\mathcal{R}) \mid a \subseteq r_i\}|$. Using $|\mathcal{UC}_i(a)| \leq m_{i+1} \leq m$, the overall complexity is $O(nm + m \log m \sum_i \sum_{a \in \mathcal{A}(r)} |\{r \in \mathcal{R} \mid a \subseteq r\}|) = O(nm + \overline{k} m^2 \log m)$ by definition of average overlap. The $O(nm^2)$ bound is obtained by using Algortihm 1 instead of Algorithm 2.

We now derive a bound depending on the average overlapping degree $\overline{K}$ of combinations. Consider an atom $a \in A_i$ and an uncovered combination $c \in \mathcal{UC}_i(a)$. We can associate a to an atom $a.desc \subseteq a$ in $\mathcal{A}(\mathcal{R})$. As c is also a combination in $\mathcal{C}(\mathcal{R})$, we have $c \in \mathcal{C}(a.desc)$ where $\mathcal{C}(s)$ denotes the combinations of $\mathcal{R}$ containing s. As the atoms in A_i are disjoint, the atoms *a.desc* for $a \in A_R$ are pairwise distinct. We thus have $\sum_{a \in A_i} |\mathcal{UC}_i(a)| \leq \sum_{a \in \mathcal{A}(\mathcal{R}) | a \subseteq r_i} |\mathcal{C}(a)|$. The overall complexity of atom computation is $O(nm + \log m \sum_{a \in \mathcal{A}(\mathcal{R})} \sum_{c \in \mathcal{C}(a)} |\{i \mid a \subseteq r_i\}|) = O(nm + k \log m \sum_{a \in \mathcal{A}(\mathcal{R})} |\mathcal{C}(a)|) = O(nm + k\overline{K} m \log m)$ by definition of overlapping degree and average overlapping degree respectively. The refined bound in terms of $T_{set}, T_{coll}(m), T_{inter}(m), T_{updt}(m)$ is obtained similarly. □

A.3 Application to forwarding loop detection (Corollary 1)

Corollary 1 directly follows from the following claim and Theorem 1.

***Claim* 1.** Given the collection $\mathcal{R}$ of rule sets of a network $\mathcal{N}$, and for each atom $a \in \mathcal{A}(R)$ the list $\mathcal{R}(a)$ of sets in $\mathcal{R}$ that contain a, forwarding loop detection can be solved in $O(\overline{k} n_G m)$ time where $m = |\mathcal{A}(\mathcal{R})|$ is the number of header classes, $\overline{k}$ is the average overlapping degree of $\mathcal{R}$ and n_G is the number of nodes in $\mathcal{N}$.

Proof. For that, we assume that each rule set $r \in \mathcal{R}$ is associated with the list L_r of forwarding rules (r, a) that have rule set r. Each such rule is also supposed to be associated to the node u whose table contains it and the index i of the rule in $T(u)$. Each list L_s is additionally supposed to be sorted according to associated nodes. Such lists can easily be obtained by sorting the collection of all forwarding tables according to the predicate filters of rules.

The claim comes from the fact that uncovered combination in $\mathcal{UC}(\mathcal{R})$ represent atoms of $\mathcal{A}(\mathcal{R})$. It follows from testing for each header class $a \in \mathcal{A}(\mathcal{R})$ whether the graph $G_a = G_h$ for all $h \in a$ has a directed cycle. G_a is computed by merging the lists L_s for $s \in \mathcal{R}(a)$ in time $O(|\mathcal{R}(a)|\, n_G)$. This graph has at most n_G edges and cycle detection can be performed in $O(n_G)$ time. The overall complexity follows from $\overline{k}m = \sum_{a \in \mathcal{A}(\mathcal{R})} |\mathcal{R}(a)|$ by definition of $\overline{k}$. □

Appendix B : Difficult inputs for previous works

B.1 Veriflow

Verfilow [17] represents the complementary of a multi-range $r = [a_1, b_1] \times \cdots \times [a_d, b_d]$ as the union of $2d - 1$ multi-ranges (at most):

- $[0, a_1 - 1] \times H_{2..d}$ and $[b_1 + 1, \infty_1] \times H_{2..d}$,
- $[a_1, b_1] \times [0, a_2 - 1] \times H_{3..d}$ and $[a_1, b_1] \times [b_2 + 1, \infty_2] \times H_{3..d}$,
- $\cdots$,
- $[a_1, b_1] \times \cdots \times [a_{d-1}, b_{d-1}] \times [0, a_d - 1]$ and $[a_1, b_1] \times \cdots \times [a_{d-1}, b_{d-1}] \times [b_d + 1, \infty_d]$,

where ∞_i denotes the maximum possible value in field i, and $H_{i..j} = [0, \infty_i] \times \cdots \times [0, \infty_j]$ denotes the multi-range of all possible values for fields $i, \ldots, j$ for $1 \leq i \leq j \leq d$.

The difficult input for Veriflow consists in a network with $n = dp + 1$ rules associated to the following multi-ranges:

- $r_0 = H_{1..d}$,
- $r_i^j = H_{1..i-1} \times [a_j, a_j] \times [b, b]^{d-i}$ for $i, j \in [1..d] \times [1..p]$.

Consider the sub-classes generated while computing $r_0 \cap \left(\cap_{i,j \in [1..d] \times [1..p]} \overline{r_i^j}\right)$. The union of multi-ranges representing $\overline{r_i^j}$ contains in particular $H_{1..i-1} \times [0, a_j - 1] \times H_{i+1..d}$ and $H_{1..i-1} \times [a_j + 1, \infty_i] \times H_{i+1..d}$. This implies that Veriflow generates on such an input all p^d sub-classes of the form $I_1 \times \cdots \times I_d$ with $I_i = [a_j + 1, a_{j+1} - 1]$ for some $j \in [0, d]$ (we set $a_0 = -1$). Forwarding loop detection of an n_G-node network thus requires $\Omega(p^d n_G) = \Omega(\left(\frac{n}{d}\right)^d n_G \frac{m}{d})$ time for Veriflow. As this example has overlappingd degree 2, this justifies the two lower-bounds indicated for Veriflow in Table 1 for d-multi-ranges.

It is possible to adapt Veriflow to support general wildcard matching by considering each field bit as a field. The wildcard expressions $r_0 = *^\ell, r_1 = 01^{\ell-1}, \ldots, r_\ell = 0^{\ell-1}1$ will then similarly generate all $2^{\ell/2}$ sub-classes obtained by concatenation of words 10 and 11. This justifies the two lower-bounds indicated for Veriflow in Table 1 for ℓ-wildcards.

Appendix B.2 : HSA/NetPlumber approach

The NetPlumber approach could be generalized to more general types of rules. However, we show that the simple heuristic for emptiness tests described in Section 5.2 is not sufficient. We provide an example where the HSA/NetPlumber approach generates an exponential number of paths while the number of classes is linear if it relies solely on this heuristic. Consider header space $H = \{1..n\}$ and the following $n+1$ rule sets: $r_1 = \overline{\{1\}}, \dots, r_n = \overline{\{n\}}$ and $r_{n+1} = H$. Consider a network $\mathcal{N}$ with $n_G = n(n+1)$ nodes. Each node $u_{i,j}$ for $0 \leq i \leq n$ and $1 \leq j \leq n$ has table $T(u_{i,j}) = (r_{i+1}, FwdD_{i+1,1}), \dots, (r_n, FwdD_{i+1,n}), (r_{n+1}, FwdD_{i+1,n})$ where action $FwdD_{i,j}$ indicates to forward packets to node $u_{i,j}$ for $i \leq n$ and to drop packets for $i = n+1$. Starting from $u_{0,1}$, the HSA approach generates a path for each combination $r_{i_1} \cap \cdots \cap r_{i_p}$ for $p \leq n$ and $1 \leq i_1 < \cdots < i_p \leq n$. This path goes through $u_{0,1}, u_{1,i_1}, \dots, u_{p,i_p}$ and then through $u_{p+1,n}, \dots, u_{n,n}$. It is constructed at least for term $r_{i_1} \cap \cdots \cap r_{i_p} \setminus \cup_{j \notin \{i_1,\dots,i_p\}} r_j$. The heuristical emptiness test of NetPlumber does not detect that it is empty since $r_{i_1} \cap \cdots \cap r_{i_p}$ contains j for $j \notin \{i_1, \dots, i_p\}$ and it is not included in r_j. The number of paths generated is thus at least $\sum_{1 \leq p \leq n} \binom{n}{p} = 2^n - 1$. However, the header classes are all singletons of H and their number is $m = n$. Note the high overlapping degree $k = n$ of this collection of rule sets.

Appendix C : Linear fragmentation versus overlapping degree

Interestingly, a complexity analysis of HSA loop detection is given in the technical report [14] under an assumption called "linear fragmentation". This assumption, which is based on empirical observations, basically states that there exists a constant c such that a given routing path P will branch at most c times. More precisely it states that a term $s = c_0 \setminus \cup_{i=1..p} c_i$ in the expression representing H_P, the set of headers that can follow P, intersects at most c of the rule sets in the table of the end node of P. A simple induction allows to bound the number of terms generated by all paths of p hops from a given source by $c^p n$. Under linear-fragmentation, the time complexity of HSA loop detection (excluding emptiness tests) is thus proved to be $O(c^{D_G} D_G n^2 m_G)$ in [14] where D_G is the diameter of network graph G, n the number of rules, and m_G the number of ports in G (in our simplified model each node has a single input port and $m_G = n_G$ the number of nodes in G). It is then argued that in practice the constant c gets smaller as the length p of the path considered increases and that practical loop detection has complexity $O(D_G n^2 m_G)$ as claimed in [15]. However, it is not rigorous to neglect the (exponential) c^{D_G} factor under the sole linear-fragmentation hypothesis.

Additionally, we think that low overlapping degree provides a simple explanation for the phenomenon observed by Kazemian et al.: as the path length increases, the terms representing the header that can traverse the path result from the intersection of more rules and become less likely to intersect other rules when overlapping degree is limited. Moreover, bounded overlapping degree k implies that the number of terms generated by HSA within p hops is bounded by $O(n^{\min(p,k)})$. The total number of terms generated is thus bounded by $O(n_G n^k)$. This guarantees that all HSA computations besides emptiness tests remain polynomial for constant k. In contrast, with the example provided in Appendix B.2 (which has unbounded overlapping degree), the HSA approach can generate exponentially many paths compared to the number of header classes in the context of general rules.

Appendix D : Related notion of weak completeness

In the context of resolution of conflicts between rules, Boutier and Chroboczek [5] introduce the concept of *weak completeness*: a collection $\mathcal{R}$ is weakly complete iff for any sets $r, r' \in \mathcal{R}$, we have $r \cap r' = \cup_{r'' \subseteq r \cap r'} r''$. They show that this is a minimal necessary and sufficient condition for all rule conflicts to be solved when priority of rules extends inclusion (i.e. r has priority over r' when $r \subsetneq r'$). Interestingly, we can make the following connection with this work: given a combination collection $\mathcal{C}' \subseteq \mathcal{C}(\mathcal{R})$ containing $\mathcal{UC}(\mathcal{R})$, we have $a(c) = c \setminus \cup_{c' \in \mathcal{C}' | c' \subsetneq c} c'$ for all $c \in \mathcal{C}'$. (See Lemma 2 in Appendix A.2.) This allows to show that $\mathcal{UC}(\mathcal{R})$ is weakly complete. It is indeed the smallest collection of combinations of $\mathcal{R}$ that contains $\mathcal{R} \cup \{H\}$ and that is weakly complete. Our work thus also provides an algorithm for computing such an optimal "weak completion".

Appendix E : Topological header classes

Inspired by the HSA/NetPlumber [15, 13] approach, we can refine the definition of header classes. We fix a source node s. A forwarding path P originating from s is a sequence $(u_0, i_0), \ldots, (u_p, i_p)$ where $u_0, \ldots, u_p$ are the nodes encountered along the path and $i_0, \ldots, i_p$ are the indexes of the forwarding rules followed: rule i_0 is applied at node $u_0 = s$, then rule i_1 at node u_1 and so on. We then define two headers as topologically equivalent from s if they follow the same forwarding paths. Note that each topological class for that relation corresponds to a unique path (if some headers match two distinct rules at a node, the first one is followed).

We now show that the topological classes (from s) are indeed certain atoms of partial collections of rule sets. Given a forwarding path $P = (u_0, i_0), \ldots, (u_p, i_p)$. Let $\mathcal{R}^j = \{r_1^j, \ldots, r_{n_j}^j\}$ denote the collection of rule sets of node u_j where $r_1^j, \ldots, r_{n_j}^j$ corresponds to the order of the n_j rules in $T(u_j)$. Let also $\mathcal{R}_i^j = \{r_1^j, \ldots, r_i^j\}$ denote the partial collection of the i first rule sets. The set of headers that can follow rule i in node u_j is $s_i^j = \overline{r_1^j} \cap \cdots \cap \overline{r_{i-1}^j} \cap r_i^j$. If P corresponds to a topological class $cl(P)$, we have $cl(P) = s_{i_0}^0 \cap \cdots \cap s_{i_p}^p$. As the only positive terms of this intersection are $r_{i_0}, \ldots, r_{i_p}$, $cl(P)$ is indeed the atom $a(c_P)$ associated to the combination $c_P = r_{i_0}, \ldots, r_{i_p}$ in $\mathcal{A}(\mathcal{R}_{i_0}^0 \cup \cdots \cup \mathcal{R}_{i_p}^p)$. Note that we consider a different partial collection $\mathcal{R}_{i_0}^0 \cup \cdots \cup \mathcal{R}_{i_p}^p$ for each path. Our framework can thus be applied to test whether the combination c_P associated to any path P does correspond to a topological class by testing the emptiness of $a(c_P)$. We detect a forwarding loop (starting from s) as soon as a path P reaches a former node of P.

Our incremental algorithm is well suited for incrementally augmenting the collection of rules as we progress along the path. Using a persistent style implementation for the data structure storing collections (as classically done when implementing binary search trees in functional languages), we can step back at a branching node v and follow a different branch without having to recompute the collection of atoms generated by the path up to v. The collection is also incrementally augmenting as we progress in routing table of a branching node v and explore different paths. This allows to efficiently perform a search of the graph in a depth first search manner.

It is important to note that two different explored paths P and Q are associated to different combinations $c_P \neq c_Q$. To see this, suppose that P and Q branch at node u: P follows rule r_i of $T(u)$ while Q follows r_j with $i < j$. After (u, i), path Q cannot follow any rule associated with set r_i due to emptiness tests. This ensures that the number of paths generated is bounded. Similarly to the proof of Proposition 2 and Theorem 2, each combination c_P generated by a path P can be associated to an atom of $\mathcal{A}(\mathcal{R})$ where $\mathcal{R}$ denotes the collection of all rule sets in the network. The number of topological classes is thus always bounded by $m = |\mathcal{A}(\mathcal{R})|$. However, for a given collection $\mathcal{R}$ of rule sets generating m atoms, one can easily produce a network topology with m topological classes.

When performing the search from source s, the number of paths (and prefixes of paths) generated is at most $\sum_{a \in \mathcal{A}(\mathcal{R})} |\mathcal{UC}(a)| = Km \leq nm$ where $\mathcal{UC} = \mathcal{UC}(\mathcal{A})$ are the uncovered combinations generated by $\mathcal{R}$. As each path prefix accounts for one call of Algorithm 1, this search thus costs $O(nm^3)$ elementary set operations. Repeating this for all possible source nodes, we get an overall complexity of $O(n_G nm^3)$ operations for forwarding loop detection. (This complexity analysis can be refined in terms of overlapping degree and optimized by using potentially Algorithm 2 instead of Algorithm 1.)

Note that the number of paths explored could grow exponentially without appropriate emptiness tests as exemplified in Appendix B.2. Although this approach remains polynomial in terms of n and m, its complexity guaranties are less interesting than what we propose in Section 4. However, it allows to extend our framework with write actions as detailed in the next appendix.

Appendix F : Write actions

First note that allowing any write operations and variable header length (by allowing to push sub-headers) make the forwarding loop detection problem undecidable as we can then easily simulate a pushdown automaton with several stacks. However, in the context of MPLS, it is natural to allow to push and pop MPLS headers (and only that type). In the classical functioning of MPLS, each push or swap action always writes a fixed label value, rules taking into account the MPLS header always have higher priority and base their decisions on the outermost label only (and no other field). In that case, paths with MPLS forwarding can be factored out by adding shortcut edges (from the first push action to last pop action) in a preliminary step which should also include loop detection for each label value pushed somewhere in the network. This can clearly be performed in polynomial time. We reserve the study of more general write action with push and pop actions for future work. However, we now sketch how to handle write actions in the context of fixed length headers.

We suppose that a forwarding rule applied to a packet can perform a write action before forwarding the packet. We typically think of write actions that write some fixed values at fixed positions. We then see a write action as a projection of the header space in a smaller sub-space. More generally, we assume that each write action w is associated with a function $p_w : H \to H$ with the following properties (we set $p_w(s) = \{p_w(h) \mid h \in s\}$ for $s \subseteq H$) :

- intersection preservation: for $s \cap s' \neq \emptyset$, we have $p_w(s \cap s') = p_w(s) \cap p_w(s')$;
- two consecutive write operations w and w' are equivalent to a single write operation w'' with $p_{w''} = p_{w'} \circ p_w$, w'' is then called a write pattern assumed to be efficiently computable.

We do not need to formally assume that p_w is a projection (i.e. $p_w \circ p_w = p_w$) although this is typically the case.

We additionally make the following assumption with respect the data-structure $\mathcal{D}$ used for sets:

- $p_w(s)$ is in $\mathcal{D}$ and can be computed in $O(T_\ell)$ time.

All the above requirements are clearly met for any write action (of fixed bits at fixed positions) in the context of wildcard expressions. In the context of prefix matching we can only allow to write a prefix of bits to ensure that a set represented by a prefix is always mapped to a set that can be represented by a prefix. We can generalize this to the context of interval matching when low order bits are ordered first (little-endian order). We still represent a set s with an interval $[a, b]$ but we additionally consider a write pattern w and the number i of bits of w written so far. The write pattern w is simply an integer with same bit representation as a and b. To test whether a header h is in s, we first check that its i lowest bits are identical to those of w, then shift a, b, and h by i bits to erase low order bits, obtaining a', b', h' respectively and test whether $a' \leq h' \leq b'$. Other set operations are handled similarly. Note that the computation of $p_{w'}([a, b])$ just requires to update the write pattern which can be globally shared for a collection of sets. A similar trick can be used with wildcard expressions. In the context of multi-ranges, our model thus enables prefix writes on one or several fields. (This includes in particular writing of exact values in one or several fields.)

Forwarding loop detection can then be performed using the approach detailed in Appendix E by additionally decorating each node of an explored path with a write pattern w equivalent to the sequence of write actions performed so far along the path. A forwarding loop is detected when a former node u associated to write pattern w is reached again with same write pattern $w' = w$ while growing a path P containing u. If the number the of write patterns that can be generated by combining various write actions is bounded by p, then the length of each path is at most $n_G p$.

Our algorithms for incremental atom computation can easily be extended in that context: when a write action w is performed when growing a path P, we update the collection of uncovered combinations representing the atoms of the current collection $\mathcal{R}_P$ of rule sets encountered along P as follows. Each combination c is replaced by $p_w(c)$ (which typically amounts to a single update a the global write pattern in the contexts of wildcard expressions and multi-ranges). We then recompute the inclusion relations between them and atom sizes in $O(m^2)$ elementary set operations as in Algorithm 1. Combinations that are detected as covered are removed (or just saved apart for efficient backtracking). We finally obtain a valid representation of $\{p_w(r) \mid r \in \mathcal{R}_P\}$ since p_w preserves intersection.

Note that write operations can only reduce the number of atoms. Forwarding loop detection can thus be performed within a factor $n_G p$ compared to the search procedure of Appendix E, that is in $O(n_G^2 n m^3 p)$ elementary set operations. This is again polynomial in terms of number of rules n, number of atoms m generated by the collection of rule sets, and the number p of write patterns that can be generated by write operations.