Epinoia: Intent Checker for Stateful Networks

Huazhe Wang*[1] Puneet Sharma† Faraz Ahmed† Joon-Myung Kang†[2] Chen Qian‡ Mihalis Yannakakis§
* Microsoft † Hewlett Packard Labs ‡ UC Santa Cruz § Columbia University

2021 International Conference on Computer Communications and Networks (ICCCN) | 978-1-6654-1278-0/21/$31.00 ©2021 IEEE | DOI: 10.1109/ICCCN52240.2021.9522299

***Abstract*—Intent-Based Networking (IBN) has been increasingly deployed in production enterprise networks. Automated network configuration in IBN lets operators focus on intents–i.e., the end to end business objectives–rather than spelling out details of the configurations that implement these objectives. Automation brings its own concerns as the administrators cannot rely on traditional network troubleshooting tools. This situation is further exacerbated in the case of stateful Network Functions (NFs) whose packet processing behavior depends on previously observed traffic patterns. To ensure that the network configuration and state derived from network automation matches the administrator's specified intent, we propose, Epinoia, a network intent checker for stateful networks. Epinoia relies on a unified model for NFs by leveraging the causal precedence relationships that exist between NF packet I/Os and states. Scalability of Epinoia is achieved by decomposing intents into sub-checking tasks and maintaining a causality graph between checked invariants. Epinoia checks for network-wide intent violations incrementally to reduce overhead in the event of network changes. Our evaluation results using real-world network topologies show that Epinoia can perform comprehensive checking within a few seconds per network with intent updates.**

I. Introduction

The increasing complexity of business rules and policies coupled with the increasing size of today's networks has made the tasks of network administrators extremely difficult. In particular, the manual conversion of network-wide business objectives to network configurations can be error-prone and difficult to troubleshoot. The advent of software-defined principles for network management has led to Intent-Based Networking (IBN) [1]. IBN aims to make networks more reliable and efficient by automatically converting network-wide objectives, called *intents* (e.g., all critical services in the data center are available to remote sites) into detailed network configurations that implement those intents. While IBN eases the configuration task for network administrators, it faces several challenges. The first challenge is handling undetected bugs and inaccuracies in the automation logic itself given that dealing with the diversity of network devices and services effectively is hard. The second challenge is the subjective nature of *intents*, which cannot be completely fulfilled by automation and might need human intervention to provide input or make changes that are not supported by the automation framework. Furthermore, network configurations need to be continuously changed to serve the ever evolving business requirements and to address security and performance issues. According to a study from Google, 70% of network failures occurred when changing network configurations [13]. Thus, a fundamental requirement of an IBN system is the ability to ensure that an administrator's intents and expectations are met through the inevitable changes and transformations in the network. In order to address this requirement, recent work has focused on network verification to guarantee that the configurations generated by automation or humans are correct (e.g., no users at remote sites should lose connectivity to the data center after being inserted to an Access Control List (ACL)).

Stateful networks refer to the networks that contain stateful Network Functions (NFs). Compared with legacy switches and routers, NFs implement more diverse functions and their packet processing behavior may depend on the packet history previously encountered. Examples of stateful NFs include firewalls that allow inbound packets if they belong to established connections and web proxies that cache popular content etc. NFs are becoming increasingly prevalent in today's network, further aggravating the problems encountered managing intent-based networks: for instance, 43% of network intent violations involve NFs, and between 4% and 15% of them are the result of NF misconfiguration [18]. However, recent work on network verification either only ensures correct NF traversal assuming all instances of each type of NFs are equally and correctly configured [10] [8] [4], or only checks NF configurations in a restricted scope that may lose end-to-end expressiveness and accuracy [17]. We have identified three key requirements of an intent checking system for stateful networks: 1) Vendor-agnostic model specifications to support diverse NFs and their configurations from different vendors, 2) Completeness to support end-to-end intent checking, to handle packet header modifications by NFs and routing dynamics, and 3) Incremental checking to efficiently check the correctness to avoid performing full checking for every change.

Existing network verification work mostly consists of two approaches: The **customized approaches**, such as HSA [10] and its real-time version NetPlumber [9], identify the set of packets affected by the network changes and utilize customized path-based algorithms to calculate their new forwarding paths. This approach is unable to model extra packet sequences from other parts of the network and thus cannot be used for stateful networks. The **solver-based approaches**, such as Minesweeper [4] and VMN [17], encode all possible packet behavior within the network using first-order logic; To achieve scalability with modern solvers, such as SAT [14] and SMT (Satisfiability Modulo Theories) [6], they rely on optimizations to identify logically independent network slices. However, there is no guarantee that these slices always have moderate size or even exist, especially when there are NFs that

[1]This work was completed while Huazhe Wang was an intern at Hewlett Packard Labs

[2]Now at Google

978-1-6654-1278-0/21/$31.00 ©2021 IEEE

TABLE I
EPINOIA VS. OTHER NETWORK VERIFICATION TOOLS
○UNSUPPORTED ◐PARTIAL SUPPORT ●SUPPORT

	Vendor-agnostic NF models	Header transformation	Incremental checking
HSA/NetPlumber [10] [9]	○	●	●
Minesweeper [4]	◐	○	○
VMN [17]	●	◐	○
Epinoia	●	●	●

modify packet headers. Further, both Minesweeper and VMN solve all constraints as a whole, and cannot reuse previous checking results when the network changes.

In this paper, we present Epinoia, an intent checker for stateful networks. Table I shows a comparison of Epinoia with other related works in terms of support for the key requirements described above. To the best of our knowledge, Epinoia is the only system that can fully support all the key requirements. This paper makes following contributions:

- A novel configuration model for NF function units represented as vendor-agnostic extensions of OpenConfig YANG models [15] that can be combined to represent configurations of commercial advanced NFs [16]. Proposes new techniques leveraging causality precedence relationships [21] between packet I/Os and NF states to represent stateful NF operating logic.
- The design and implementation of a scalable yet correct approach for intent checking based on intent decomposition and incremental checking using a novel causality graph memoization technique for all checked results.
- A comprehensive evaluation of Epinoia using a real-world dataset and topologies. Epinoia can perform incremental checking within a few seconds per network and/or intent update which reduces the time cost by up to a factor of 100x compared with a full checking for all intents.

II. EPINOIA DESIGN AND ARCHITECTURE

Consider the network pictured in Figure 1 with an end host subnet S_0 and a server subnet S_1. FW_1 and FW_2 are two stateful firewalls and PY is a forward proxy that works as an intermediate agent between clients and servers. The operators intend to block traffic from S_0 to S_1. The bottom of Figure 1 shows configuration snippets that implement this intent. Line 1 is a security rule at FW_1 that denies all packets from S_0 to S_1. However, as FW_1 conducts stateful processing, those packets may still be allowed if they belong to established connections initiated from S_1. To prevent such connections to be established, a similar deny rule for packets from S_1 to S_0 (line 3) is added at FW_2. Even with this simple example, checking intent using existing tools could give inaccurate results and be time-consuming.

Static vs. temporal modeling. Recent work on network control plane configuration (e.g., BGP configuration) synthesis [7] and verification [4] have shown that route advertisements between routers can be effectively modeled using boolean variables. Following this idea, a packet P_0 from S_0 can reach S_1 through FW_1 and FW_2 can be represented as a boolean

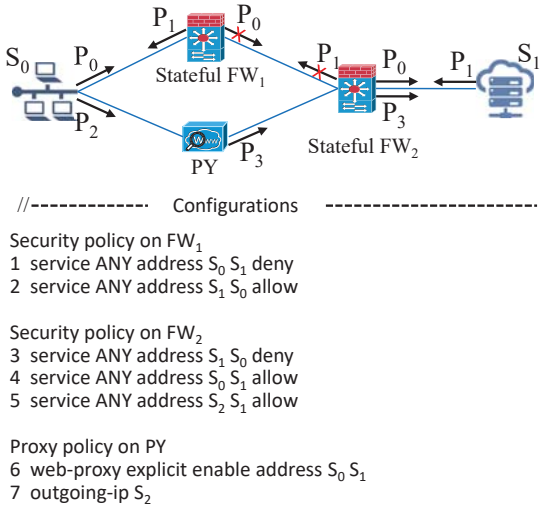

Fig. 1. Example NF configuration snippets.

variable r_0. For r_0 to be $True$ when P_0 is denied by the security rule at FW_1, an earlier reverse packet P_1 has to be allowed at FW_1 from S_1 to S_0. We denote this reverse reachability as r_1, and we have $r_0 \Rightarrow r_1$. However, due to the deny rule at FW_2, the fact that P_1 from S_1 to S_0 can reach FW_1 indicates an earlier reverse packet P_0 to be allowed at FW_2 from S_0 to S_1, denoted as $r_1 \Rightarrow r_0$. Given equations above, static analysis without temporal representation may report a violation of the block intent when both r_0 and r_1 are $True$. However, this turns out to be a false alarm. FW_1 will allow P_0 only if it saw P_1 before, which requires P_1 to be allowed by FW_2 at a first place. Thus, it cannot rely on the state created by P_0. This example shows the necessity to include temporal representation to model stateful networks as packets may have different behavior at stateful NFs when they arrive in different sequences.

Partial vs. complete path set. To scale with modern solvers, several optimization techniques have been studied in solver-based approaches [4] [17]. The core idea is to reduce the size of constraints given to the solver by restricting packet headers and their forwarding paths based on destination addresses. That is, the checking is conducted over a slice of the network (e.g., a single forwarding path). Though such simplifications could reduce the time cost, they may also lose completeness and lead to unsound checking results, especially when there are NFs that modify packet headers. One such example is shown in Figure 1. Line 6 of the configuration snippets indicates that PY in the bottom will explicitly intercept request packets from S_0 to S_1 and forward them with a new source S_2 (line 7). Those packets are also allowed at FW_2 (line 5). Instead of sending packets directly to S_1, a host in S_0 could first send packets to PY, which then forwards the packets to S_1. This indicates a potential violation of the block intent between S_0 and S_1. In addition, networks are built with fault tolerance. Critical services are multi-homed, and communication endpoints have redundant paths. The dynamic nature of the underlying routing plane may assign different paths at different time even for the same set of packets. The

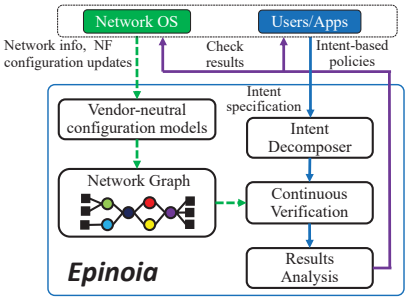

Fig. 2. Epinoia workflow

NF processing taken depends on the path a packet actually traverses. *Configurations of NFs must ensure that no potential path violates network intents.*

Host vs. group level querying. In existing intent-based systems, all intents are specified with respect to end point groups (e.g., engineering department, a group of servers) [19] [2] [3]. Recall the previous intent: S_0 should not be able to reach S_1. Consider S_0 as a guest network with 100 hosts and S_1 to be a data center with 1000 servers. To check the block intent, the naive approach of exploding the query into 100 thousand separate queries is too slow. A typical effective solution would convert the original query and check whether its negation can be satisfied. However, due to the stateful processing of NFs, this technique cannot be applied for stateful networks. More details are discussed in §IV-A. We observe that NF processing policy commonly partitions end hosts into policy equivalence groups, i.e., into set of end hosts, to which the same policy applies. In Epinoia, endpoints relating to the same set of intents are represented as groups and queries for the same group are aggregated to achieve better efficiency.

Epinoia Overview. Figure 2 illustrates an overview of the Epinoia workflow with its key components. Epinoia allows users/applications to specify network intents based on extended policy graph models (§III-A). NFs from different vendors may support different configurations and features. We break down the functionalities of advanced NFs into function units and propose vendor-neutral configuration models for each function unit (§III-B1). Such function units can be combined and extended to support real-world NFs. To correlate configurations of NFs and packet behavior in stateful networks, we formulate key causal precedence relationships [21] among NF packet I/Os and states (§III-B3). All constraints are attached to a network graph, containing all potential paths needed to be checked for each intent to ensure that NF configurations match intents under arbitrary routing dynamics. Along each path, an end to end intent is decomposed into sub checking tasks (§IV). Each smaller task can be efficiently checked using a SMT solver. The continuous verification module maintains a causality graph with all checked results (§V). The goal is to enable the intent checker to check for network-wide intent violations incrementally whenever there are changes to network and/or intent. Finally, checking results are analyzed and all reported violations are returned to the network OS or intent creators.

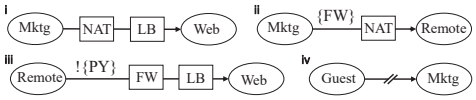

Fig. 3. Example network intents

III. Intent and Network Models

A. Network Intent Specification

Network intents specify the desired outcome of the network. In this paper, we look at two very basic intents: reachability and isolation, which can be used as building blocks to implement other advanced intents. Epinoia extends the intent specifications in PGA [19]. Our choice is motivated by the intuitive graph representation of network intents, support of NF chaining. Figure 3 shows four example network intents in an enterprise network. Nodes are pre-defined end point groups and directed edges indicate the communication intents between endpoints. Boxes along edges specify the required NF traversal for each communication. In addition to the required ones, constraints on possible optional NFs are annotated on each edge segment in the form of $\{NF_1...NF_n\}$. Similarly, avoidance of NFs are specified using the form $!\{NF_1...NF_n\}$. For an isolation intent, a double slash is added on the edge to indicate that the communication must be blocked. The four intents in Figure 3 are: i) Marketing department should be able to access web services and the traffic must go through a NAT and a load balancer. ii) They should also be able to access remote sites by going through a NAT and possibly one or more firewalls before the NAT. iii) Packets from remote sites to web services must be inspected by a firewall and a load balancer. No proxy is allowed before they are inspected by any firewall. iv) Packets from guest networks to the marketing department must be blocked.

B. Network Models

1) NF Configuration Models: Recent work on NF modeling has shown that NFs of the same type from different vendors have similar operating logic [27] [24] [17]. For example, the firewall function of iptables [20], pfSense [22] as well as Palo Alto Firewall [16] all start with detecting whether a packet belongs or relates to an established connection. Then the packet is matched against a list of ACLs. If one is found and it allows the packet, then the packet is forwarded; otherwise it is dropped. Contrary to the similarity in the operating logic, we observe that NFs differ greatly in the format or features they support in their configurations, which are the main inputs that operators provide and want to check before they are installed into NFs. To mitigate the complexity brought by vendor specificity, open source communities such as OpenConfig [15] as well as some emerging IBN platforms in industry (e.g., Apstra AOS [3] and Google Zero Touch Network [13]) have been working on designing vendor-neutral configuration models. However, most of those models are for routers or routing related protocols and none include NFs. Another observation is that advanced NFs usually consist of a chain of basic functions. For example, a Palo Alto Firewall can be configured

to implement a firewall-NAT-Load balancer chain. Inspired by the observations above, in Epinoia, we have proposed vendor-agnostic configuration models for common function units (e.g., address objects, security rules, NAT rules) which are written as extensions of the OpenConfig YANG models. Models for each function unit can be combined to form the configuration model of more advanced NFs. Moreover, with off-the-shelf tools, configurations written using the model can be easily converted into serialization formats (e.g., JSON) for other services (e.g., network intent verification). Listing 1 shows an example configuration instance of a real security rule in JSON. The model is extensible to support additional features based on the actual functionalities of NFs.

```
...     "security-rules": {
          "security-rule": {
            "23": {
              "id": "23",
              "config": {
                "src-address": "guest",
                "dst-address": "marketing",
                "service": "ANY",
...             "action": "DENY",
```

Listing 1. Snippets of a security rule in JSON

2) Network Graph: To obtain the complete path set that should be checked for intents, Epinoia models a network as an undirected graph. Nodes in the graph are either endpoints or NFs while edges represent possible packet exchanges between those nodes. Such a graph can be extracted by traversing the network topology: if the current node is a switch or has been visited, continue to examine the next node; otherwise, create a new node in the network graph representing the corresponding NF or endpoints. Note that Epinoia does not aim to check the correctness of stateless switching fabrics as there already exist plenty of solutions [9]–[11], [28]. Meanwhile, by removing the switching fabric, the network graphs result in much smaller sizes (degrade the size by at least 50% [23]) but are still able to capture all potential paths.

Figure 4 shows a network graph of an example network. Internal endpoints m_1 and g_1 belong to the marketing and guest networks, connected to remote sites with two firewalls. A web service is hosted in a demilitarized zone, guarded by a destination NAT and a load balancer. *Epinoia leverages an off-line path generation step to obtain all simple paths with only NFs and endpoints.* For most scenarios, the set of paths is fairly static and can be precomputed.

3) Encoding NF packet processing: The functionality of a NF can be factored into two generic parts: i) a classifier that searches for a matching over packet header fields or payload, and ii) a transfer function that transforms incoming and outgoing packets. Upon receiving a packet, based on configurations, a NF processes the packet with the actions corresponding to the rules or states that the packet matches. Naturally, the input packet on which an output depends must be received before the output is produced. In other words, there exists a causal precedence relationship [21] between the input and output. We can generically express this relationship as $send_{p_2} \Rightarrow recv_{p_1}$, where $[A] \Rightarrow [B]$ denotes event A depends on B. p_1 and p_2 correspond to the same packet before and

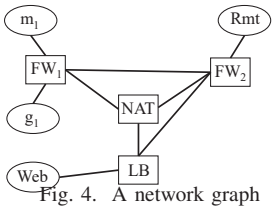

Fig. 4. A network graph

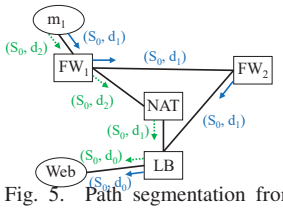

Fig. 5. Path segmentation from marketing to Web

after NF processing. Both p_1 and p_2 are subject to certain constraints determined by NF configurations. States at NFs correspond to packet histories. For example, if a content c is cached at a proxy, the proxy must have received a request packet for c and a response packet from the server that holds c before it can be cached at the proxy. Written generically: $state_s \Rightarrow recv_P$, where P represents a sequence of packets required to establish state s. Such causality also exist between one NF's output and another NF's input. For example, a packet must be sent out before it is received. Written generically: $recv_p \Rightarrow send_p$. A rich set of causalities exists in NFs, e.g., a timeout must be reached before a state expires; a configuration must be loaded before it can be applied to packets. However, most of these causalities are orthogonal to our intents. We therefore only consider packet processing causalities that affect how packets are forwarded or modified.

To encode the causal precedence relationship to a format that can be accepted by a SMT solver, it is intuitive to model packet behavior at NFs using two Boolean valued uninterpreted functions with universal/existential quantifiers. For example, we define `send(n,i,p,t)` as a sending event of packet `p` by NF `n` through interface `i` at time `t`. Similarly, receiving a packet is denoted as `recv(n,i,p,t)`. We aggregate all interfaces of a NF into either the internal (`i==0`) or external (`i==1`) interface as some NFs may apply different processing policies for inbound and outbound packets. The send and receive functions return $True$ when the input arguments correspond to a valid event in the network; or they must return $False$. We show how to capture causal precedence relationships using example SMT encodings for some common stateful NFs.

Stateful firewall. A stateful firewall (Listing 2) utilizes ACLs to determine whether to allow or deny a packet from a new connection. ACLs can be modeled using a predicate $acl\_func(a_1, a_2)$, where a_1 and a_2 correspond to the source and destination address of a packet. Packets that belong to established connections are allowed by a stateful firewall even if they are denied by ACLs. An established state indicates that the firewall has received and allowed a reverse packet before.

$$\begin{array}{l} \text{Forall } [i_0, p, t_0]\ send(fw, i_0, p, t_0)\ \text{Implies} \\ \text{Exists } [i_1, t_1]\ recv(fw, i_1, p, t_1) \wedge\ t_1 < t_0\ \wedge\ i_0 \neq i_1 \\ \\ \text{Forall } [i_0, p_0, t_0] \\ send(fw, i_0, p_0, t_0) \wedge \neg\ acl\_func(p_0.src, p_0.dst)\ \text{Implies} \\ \text{Exists } [i_1, p_1, t_1]\ recv(fw, i_1, p_1, t_1) \wedge\ t_1 < t_0\ \wedge\ i_0 \neq i_1 \wedge \\ acl\_func(p_1.src, p_1.dst)\ \wedge\ p_1 == p_0.reverse \end{array}$$

Listing 2. Encoding of a stateful firewall

Load balancer. A load balancer (Listing 3) holds a shared address ($share\_addr(a)$) for a back-end server pool

($server\_addr(a)$). Requests sent to the load balancer are randomly distributed to one of the servers and replies from servers look for a matched request which is sent back by the load balancer.

```
Forall [p0, t0] send(lb, 1, p0, t0) Implies
Exists [p1, t1] recv(lb, 0, p1, t1) ∧ t1<t0 ∧ share_addr(p1.dst)
 ∧ p1.src == p0.src

Forall [p0, t0] send(lb, 0, p0, t0) Implies
Exists [p1, p2, t1, t2] recv(lb, 1, p1, t1) ∧ recv(lb, 0, p2, t2) ∧
t2<t1<t0 ∧ p2 == p0.reverse ∧
share_addr(p2.dst) ∧ server_addr(p1.src) ∧
p0.dst == p1.dst == p2.src
```

Listing 3. Encoding of a load balancer

Reverse proxy. A reverse proxy (Listing 4) is configured with ACLs specifying which clients have access to content originating at certain servers. Upon receiving a request that is allowed by ACLs, it initiate a new request to the corresponding server if the contents have not been cached. When receiving responses from the server, it forwards the response to the client who originally requested the content.

```
Forall [p0, t0] send(py, 1, p0, t0) Implies
Exists [p1, t1] recv(py, 0, p1, t1) ∧ t1<t0 ∧ p0.src == py ∧
acl_func(p1.src, p1.dst) ∧ p0.payload == p1.payload

Forall [p0, t0] send(py, 0, p0, t0) Implies
Exists [p1, p2t1, t2] recv(py, 1, p1, t1) ∧ recv(py, 0, p2, t2) ∧
t2<t1<t0 ∧ acl_func(p2.src, p2.dst) ∧ acl_func(p0.dst, p0.src)
p1.dst == py ∧ p0.src == p1.src == p2.dst ∧
p0.payload == p1.payload == p2.payload
```

Listing 4. Encoding of a reverse proxy

IV. Intent Decomposer

Given a network intent, we can use SMT solver to check whether the intent is satisfied. However, even with the smallest network (18 nodes) we use in our evaluation, the solver cannot return an answer in a reasonable time. To improve scalability, one key observation is that though a network intent specifies a high level end to end objective, it is possible to decompose it into several sub-tasks, where each task can be checked separately. Next we present how the intent decomposer of Epinoia decomposes network intents in two dimensions.

A. Atomic Address Object

The concept of address objects (mostly referred as zones or aliases) are widely used in network management ecosystems. Assume we are about to configure a set of security rules guarding the servers in a data center to allow traffic from hosts in the marketing department while blocking mobile devices connected to the guest network. Instead of spelling out each address explicitly when a rule is added, we can define address objects as placeholders (e.g., data center, marketing department, guest network); each rule can be applied directly to such address objects. We define the set of atomic address objects which specifies the largest common refinement over the address space given the set of address objects.

To illustrate the idea of atomic address object, we represent three address objects p_1, p_2 and p_3 as ranges and place them into the address space in Figure 6. p_3 has two ranges as it specifies two non-continuous subnets. There are six non-overlapping intervals $I_0 \sim I_6$ formed by each consecutive pair

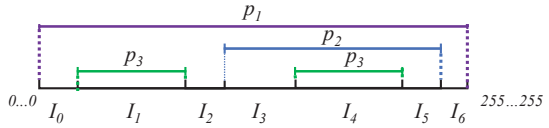

Fig. 6. Calculating the set of atomic address object for three address objects p_1, p_2 and p_3

of endpoints. The set of atomic address objects can be easily calculated by combining intervals that belong to the same set of address objects. For example, I_1 and I_4 are two separate atomic address objects. $I_0 \cup I_2 \cup I_6$ and $I_3 \cup I_5$ are the other two atomic address objects. In addition, an address object can be represented as a union of a subset of atomic address objects. For example, $p_2 = I_3 \cup I_4 \cup I_5$. We call packets sent from one atomic address object to another atomic address object as a *traffic class*. With the same network state, packets within the same traffic class are treated equally at all NFs in the entire network as they match the same set of processing rules. An endpoint group in an intent can be represented as a union of atomic address objects whose intersection with the endpoint group is not empty. To check an intent between two endpoint groups, instead of querying each pair of end hosts, we can instead simply check the more compact traffic classes between the two endpoint groups. For example, an intent from endpoint group e_0 to e_1 can be checked using two traffic classes (s_0, d_0) and (s_1, d_0) if $e_0 \cap s_{0,1} \neq \oslash$, $e_0 \subset s_0 \cup s_1$, $e_1 \cap d_0 \neq \oslash$ and $e_1 \subset d_0$. The benefit is two-fold:

Header matching elimination. Most NFs decide processing actions for incoming packets by matching packet headers against processing rules. The natural way to represent a packet and a processing rule for this check is to use bit vectors and check for equality using a bit mask. However, bit vectors are expensive and solvers typically convert them to SAT. In Epinoia, the matching fields of processing rules are represented as a set of integer identifiers for atomic address objects. Header matching at NFs are converted to integer membership check which is more efficient for solvers. For processing rules that modify packet headers (e.g., NAT rule), the modified addresses are also represented as one or more atomic address objects. Depending on a deterministic or nondeterministic modification, an incoming atomic address object is mapped to another atomic address object.

Adapting to temporal modeling. A solver usually returns a single solution when the set of constraints are satisfiable. Sometimes, we need all solutions for a query, i.e., all hosts in the marketing department should be able to reach the web service. In static modeling, this problem can be solved by testing the satisfiability of the negation of the query, However, with the temporal modeling required by stateful NFs, the negation of the query can be satisfied either with a packet that would be blocked in the network, or a packet sequence that could not have existed because it violates the casual precedence constraints. We need to differentiate between these, and find only true packet loss. To do this, we can only check an intent directly, which could boil down to a large of number of sub-queries corresponding to each pair of end hosts specified

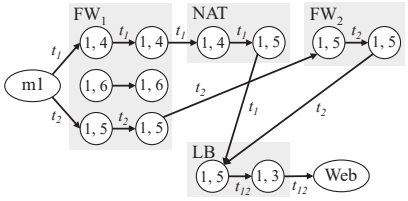

Fig. 7. The causality graph for the reachability between m1 and Web.

in the intent. With atomic address objects, the number of necessary queries as well as the total time cost is significantly reduced as the checking results can be applied to all end hosts that belong to the same atomic address object.

B. Path Segmentation

Epinoia pre-calculates all paths for each intent and an intent is satisfied if there is no violation along all potential paths. Along a path, checking an end to end intent can be divided into several sub-tasks, each task includes a single NF. The intuition is based on two observations: i) Many NFs have concrete constraints on headers of incoming or outgoing packets. For example, a source NAT translates private addresses to its public addresses; A load balancer uniformly distributes packets heading to its virtual address to a set of dynamic addresses. Such concrete constraints are specified in NF configurations and can be propagated along the path, which helps remove redundant information that the SMT solver might otherwise have to discover by itself. ii) State constraints refer to the local packet processing history at a NF. To check if a state could be valid, only constraints within the NF need to be included.

We review the intent (i) in Figure 3 within the network graph shown in Figure 4. Two potential paths from m_1 to Web are shown in Figure 5. Address pairs annotated on each path segment specify the concrete constraints on source and destination addresses of packets that can reach this segment. s_0 denotes the atomic address object corresponds to m_1 while d_0 represents Web. For packets going through FW_1, FW_2 and LB, the source address of packets are always s_0 since no NF along the path modifies the source address. For the last hop, the destination address must be d_0. As a load balancer requires an incoming packet to use its shared address as the destination address, denoted as d_1, the first three segments all have d_1 as destination address. For packets going through FW_1, NAT and LB, the source address is always s_0 while the destination address is modified from d_2 to d_1 and d_1 to d_0 at NAT and LB respectively. To check the reachability intent between m_1 and Web, Epinoia starts with checking whether those concrete and state constraints within a segment can be satisfied using a solver. A path can be valid only if all segments are satisfiable; otherwise the path is not valid.

V. Continuous verification

After checking each segment, Epinoia still needs to combine the results returned by the solver to make sure they are consistent with each other. Meanwhile, upon a network change, Epinoia should be able to identify the affected parts that may need to be rechecked. To achieve these goals, Epinoia maintains a customized causality graph that stores all checked results. Intent checking can be conducted incrementally by traversing the causality graph.

A. Causality Graph

A node in a causality graph represents either a packet sending or receiving event. Each node is tagged with a pair of atomic address objects specifying the set of source and destination addresses of the packets. An arrow in the graph indicates a causal precedence relationship among two events. The event on the front end depends on and must happen after the event on the rear end. For a single NF, it is straightforward to construct a causality graph of packet sending or receiving events required by the satisfiability assignment from the solver. When there is more than one NF, receiving a packet must be traced back along the selected path to a packet sending node. If the corresponding sending node already exists, an arrow is added between the sending and the receiving node. If not, the packet sending is checked within the upward NF and other nodes or edges are added as needed. This procedure continues until the packet receiving node is traced back to an endpoint.

Figure 7 shows an example causality graph for the two potential paths in Figure 5. Atomic address objects are represented as integers. 1 and 3 correspond to m_1 and Web respectively; 5 is the virtual address configured at the load balancer; the NAT maintains two deterministic atomic address object mapping: from 4 to 5 and 6 to 7. Consider the $FW_1 - NAT - LB$ path, possible packets received and forwarded by FW_1 are $(1, 4)$ and $(1, 6)$ since NAT only accepts packets heading to 4 and 6. We assume both packets are allowed by FW_1. Later, only packet $(1, 4)$ goes through NAT as the transformed packet must be $(1, 5)$ to be processed by LB. At LB, packet $(1, 5)$ is changed to $(1, 3)$ and finally sent to Web. Similarly, we add nodes and edges for path $FW_1 - FW_2 - LB$. We add tag t_i along each edge to identify path i. Based on the causal relationship, it's obvious that a path i is valid if the subgraph tagged by t_i has no loop, which indicates that there exists a valid time sequence for all packet sending and receiving events to achieve the end to end intent. In this example, both paths 1 and 2 are valid. To reuse the checked results, both satisfied and unsatisfied checking (not shown for simplicity) results are stored in the graph. In Epinoia, only one causality graph is maintained as the checked results can be shared among paths and intents. When the graph is storing more results, the size of a sub-graph tagged by a path identifier is independent of the complexity of the causality graph. As events occur to the network, Epinoia identifies affected intents and incrementally updates the causality graph. We handle the following six events.

Adding an address object. When a new address object is added, an existing atomic address object may be divided into two new ones. Nodes and edges related to the atomic address object should be duplicated to reflect the changes. However, an intent needs rechecking only if a new rule using the new address object is inserted.

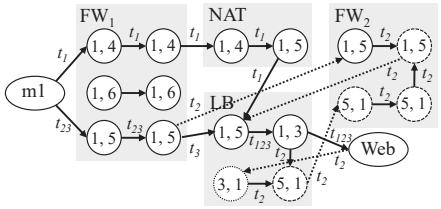

Fig. 8. The causality graph under a rule insertion and a link up.

Deleting an existing address object. Similarly, when an address object is deleted, two existing atomic address objects may specify the same atomic address object. Duplicated nodes and edges in the causality graph are removed. No intent needs to be rechecked.

Inserting a rule. To identify the set of intents that may be affected by the new rule, each node in causality graph maintains a set of intents and corresponding paths relying on the node. For example, the packet receiving node $(1, 5)$ in FW_2 is created by intent (i) in Figure 3 along path $FW_1 - FW_2 - LB$. When a new rule is inserted at a NF, Epinoia first identifies existing packet receiving nodes which will be processed by the new rule. Its set of intents must be rechecked. For other intents going through the NF, while all previous satisfied intents should not be affected as their nodes do not match the new rule, all unsatisfied intents should be rechecked. We show how the causality graph is updated when a deny rule for packet $(1, 5)$ is added at FW_2 in Figure 8. Now packet sending $(1, 5)$ requires a previous sending of $(5, 1)$, which then is traced back to a sending $(5, 1)$ at LB. At LB, the packet sending $(5, 1)$ relies on a previous sending of $(1, 3)$, which is traced back to a receiving and sending of $(1, 5)$ at LB and FW_2 respectively. After adding all necessary nodes and edges, the subgraph tagged by t_2 introduces a loop, so path 2 becomes invalid. Edges only tagged by t_2 are removed from the causality graph (dotted lines).

Deleting a rule. When a rule is deleted, intents relying on the packet receiving matching the deleted rule need to be rechecked as they will be handled by lower priority rules, and may result in different checking results.

Link up. A link up may lead to two cases where the graph needs to be updated. For each intent, Epinoia first extracts new paths from the pre-calculated path set that traverses the new link and checks if the paths are valid. Meanwhile, Epinoia checks whether packet receiving previously cannot be traced back to endpoints at the two NFs connected by the new link become valid. If so, the set of paths relying on those packet receiving events may become valid. As shown in Figure 8, if a link is up between FW_1 and LB, a new path 3 is added by going through FW_1 and LB.

Link down. When a link goes down, all the edges using that link are deleted, which in turn removes all the paths going through those edges.

B. Running Intent Checking Queries

Given an intent, Epinoia divides the intent into sub checking tasks using the intent decomposer. With the checking results

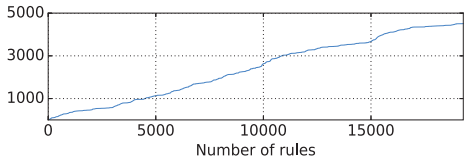

Fig. 9. Number of atomic address object as number of rules increases.

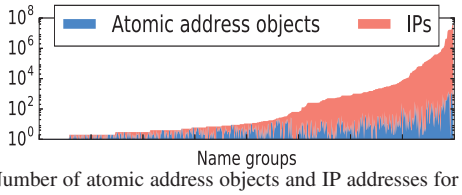

Fig. 10. Number of atomic address objects and IP addresses for name groups.

maintained by the causality graph, Epinoia calls a SMT solver only when a sub-task has not been checked before. For a reachability intent, valid paths are collected for each traffic class. Each valid path corresponds to a sequence of NFs in the network. Epinoia finds all valid paths that satisfy the NF chaining requirement in an intent. The remaining valid paths correspond to the ones that are reachable but violate the NF traversal requirements. For a block intent, any valid path indicates a potential intent violation.

Once an intent is added, it is evaluated against all future snapshots of the network graph. For all reported violations, Epinoia reports corresponding network elements or paths the violating traffic is taking. Each piece of configuration is tagged with its intent. Given a reported violation, the tag helps trace back to the intent that generates the configuration.

VI. Evaluation

We have developed a prototype of Epinoia mostly using Python. To evaluate Epinoia, we first examine how it deals with a real-world enterprise ACL dataset and then investigate the effectiveness of the intent decomposer. Finally we evaluate the runtime performance of Epinoia. All our experiments were done on a machine with 4 cores, 2.93 GHz Intel Xeon Processor and 6 GB RAM. We report times taken when the checking is performed using a single core. We use a SMT solver Z3 [5] for our evaluations. SMT solvers rely on randomized search algorithms, and their performance can vary widely across runs. The results reported are generated from 100 runs of each experiment.

A. Real-world evaluation

We obtain an ACL dataset from a policy management system of a large enterprise network. These policies are specified using 801 pre-defined address objects located at 137 compartments (groups of subsets). Each ACL rule permits or denies the communication between two address objects, each address object corresponds to one or more IP subnets (address objects may overlap with each other). Given a set of ACLs, we calculate the number of atomic address objects based on the address objects used by those ACLs. As shown in Figure 9, the number of atomic address objects increases with a slope less than $1/3$ with increased rule set size. This indicates the similarity between rules with respect to their target address

space. In total, there are over 19K ACL rules and 4508 atomic address objects. While some atomic address objects contain large address blocks, about half (2510) of them specify only a single IP. The size of the address objects also varies widely, ranging from a single IP to over 600 non-contiguous subnets (representing $\sim$ 100 million IPs). In contrast, the variation in the number of atomic address objects within an address object is much smaller. As shown in Figure 10, address objects are sorted by the number of IPs within the object. Over 90% of address objects have less than 6 atomic address objects. With fewer atomic address objects, it's more likely for Epinoia to achieve better performance when checking group level intents.

Next we use Epinoia to detect potential security breaches that may occur using the ACL dataset. We assume all compartments are connected with a full mesh topology and the ACL policies conduct stateful processing. We measure the time cost to check the reachability for each traffic class between two compartments. The average cost is 0.78 seconds with a maximum of 3.32 seconds. In total, we found 351 potential breaches due to inconsistent deny rules. For example, a packet matches a deny rule either at the local or the remote compartment, which indicates a block intent from the administrator. However, the block intent may be violated if its reverse traffic is able to pass the compartment.

B. Scalability

To evaluate the scalability of Epinoia, we quantify the effectiveness of the intent decomposer by measuring the time cost of an end to end reachability query. We connect two end hosts with a single firewall. Then we keep inserting ACL rules into the firewall and measure the time cost to check the reachability between the two hosts.

First, we represent addresses as bit vectors (BV) in the SMT encoding and use it as a baseline to show the effectiveness when atomic address objects (AA) are used. Figure 11 shows that the query time cost increases exponentially for BV based encoding while all queries cost less than one second when atomic address objects are used. This speeds up intent checking by 100x when there are 30 rules. The reason is that BV are expensive for SMT solvers and each rule inserted introduces at least 32 extra variables. However, by aggregating addresses to atomic address objects, symbolic variables representing IP prefixes are replaced with integers. A satisfied query requires more time as it needs to calculate valid assignments for all variables in the constraint set, while an unsatisfied query returns immediately when a conflict is found.

To evaluate the benefit of path segmentation, we add additional firewalls between the two hosts to create a firewall chain. We measure the time cost to check the reachability between the two hosts when all the constraints along the path are solved as a whole. This corresponds to a key optimization in VMN [17], where the checking is restricted to the forwarding path between end hosts. When the path segmentation (PS) is applied, we check each firewall one by one and sum up the time cost. As shown in Figure 12, when the path is checked as a whole, the time cost increases significantly with increased number of firewalls. The SMT solver Z3 we used in our experiments cannot return before timeout when the number of firewalls is larger than 9 for satisfied queries and 10 for unsatisfied query. With path segmentation, the time cost increases linearly and the maximum cost for satisfied query is 7.73 seconds. For unsatisfied queries, the cost does not necessarily go up with increased number of NFs as the checking process terminates whenever one of the segments cannot be satisfied. The maximum time cost is 0.26 seconds, which highlights the effectiveness of the intent decomposer in Epinoia for large networks.

C. Runtime performance

In this set of experiments, we evaluate the runtime performance of Epinoia using four topologies from Topology Zoo [12] with number of nodes ranging from 18 to 93. In our experiments, we create 200 network intents, each of which contains 0 to 10 NFs of different types and we randomly attach end hosts belonging to pre-defined address objects to different nodes in the topology. We also randomly assign a NF instance to each node in the topology. Epinoia executes a pre-computation procedure to enumerate the paths for all the intents, which could be costly for large topologies. However, we emphasize that this procedure only needs to be done once and this can be performed off-line.

In the first experiment, we check each intent one after another, and all checked results are stored in the causality graph. Figure 13 shows the cumulative time cost to check all intents for the four networks. All time costs grow slightly as the number of polices increases. The reason is that many intents share the same set of sub checking tasks for different traffic classes. The checked results can be reused among intents when there are no network changes.

With all the checked results, we next evaluate how Epinoia reacts to network dynamics. We randomly choose to insert/delete a rule or add/remove a link and measure the time cost for Epinoia to identify and recheck the set of affected intents for each scenario. As each network change may affect a different amount of intents, we report both the average and maximum time cost to recheck the affected intents in each network. As shown in Figure 14, the average cost of rechecking after a change is less than 10 seconds, with the maximum for inserting a rule in Internode being close to 20 seconds. Without the incremental checking, a full check is required for all intents whenever there is any change. The average speedup of Epinoia incremental checking is 34x, 79x, 94x and 101x for each network respectively.

VII. Related Work

To model stateful NFs, existing approaches either work on extracting models by analyzing NF source code [27] [24] [26] or hand crafted models [17] based on expert knowledge. We take a different approach, in which we have designed vendor-agnostic NF configuration models and construct NF forwarding models using key causality relationships. There is a rich body of work for verifying forwarding behaviors in

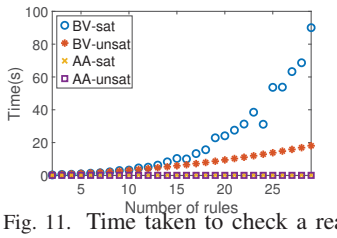

Fig. 11. Time taken to check a reachability query as # of rules increases.

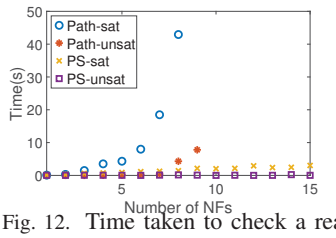

Fig. 12. Time taken to check a reachability query as # of NFs increases.

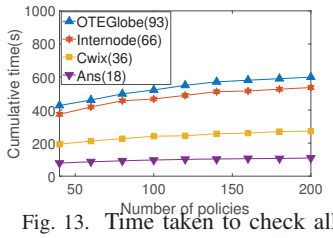

Fig. 13. Time taken to check all intents

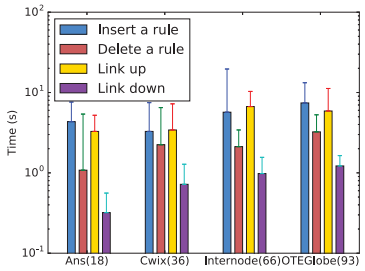

Fig. 14. Time taken to recheck affected intents per network change.

stateless networks [10] [11] [9] [28]. While these work can efficiently check a number of policies such as reachability and loop freedom, it is nontrivial to extend these work to support stateful data planes. There are several proposals on verifying network control planes [4] [8], where the processing is stateful; however, all of those work rely on a converged routing state and cannot be used for stateful NFs. To check stateful networks, Symnet [24] runs symbolic execution over an abstracted NF implementation and SFC-Checker [25] extends the network graph in HSA [10] by adding nodes for each NF state. Both of these approaches are path-based and cannot check state consistency between different NFs. VMN [17] also uses a SMT solver and identifies an end to end slice for each checking. However, VMN only supports block intents and cannot scale to large networks with dynamic updates.

VIII. Conclusions

Our intent checking solution, Epinoia, efficiently supports stateful networks with a variety of network functions. Epinoia includes vendor-agnostic network function modeling combined with capturing causality precedence relationships for incremental intent checking. A comprehensive evaluation shows that Epinoia can check network intents in under 10 seconds per network update and reduce checking time by a factor of up to 100x compared with a full checking for all intents.

References

[1] Intent based networking. https://www.cisco.com/c/en/us/solutions/intent-based-networking.html.

[2] A. Abhashkumar, J.-M. Kang, S. Banerjee, A. Akella, Y. Zhang, and W. Wu. Supporting diverse dynamic intent-based policies using janus. In *Proc. of ACM CoNEXT*, 2017.

[3] Apstra. AOS: How it works. http://www.apstra.com/products/how-it-works/. Online; accessed 2 July 2018.

[4] R. Beckett, A. Gupta, R. Mahajan, and D. Walker. A general approach to network configuration verification. In *Proc. of ACM SIGCOMM*, 2017.

[5] L. De Moura and N. Bjørner. Z3: An efficient SMT solver. *Tools and Algorithms for the Construction and Analysis of Systems*, 2008.

[6] L. De Moura and N. Bjørner. Satisfiability modulo theories: introduction and applications. *Communications of the ACM*, 54(9):69–77, 2011.

[7] A. Gember-Jacobson, A. Akella, R. Mahajan, and H. H. Liu. Automatically repairing network control planes using an abstract representation. In *Proc. of ACM SOSP*, 2017.

[8] A. Gember-Jacobson, R. Viswanathan, A. Akella, and R. Mahajan. Fast control plane analysis using an abstract representation. In *Proc. of ACM SIGCOMM*, 2016.

[9] P. Kazemian, M. Chan, H. Zeng, G. Varghese, N. McKeown, and S. Whyte. Real time network policy checking using header space analysis. In *Proc. of USENIX NSDI*, 2013.

[10] P. Kazemian, G. Varghese, and N. McKeown. Header space analysis: Static checking for networks. In *Proc. of USENIX NSDI*, 2012.

[11] A. Khurshid, W. Zhou, M. Caesar, and P. Godfrey. Veriflow: Verifying network-wide invariants in real time. *ACM SIGCOMM Computer Communication Review*, 2012.

[12] S. Knight, H. X. Nguyen, N. Falkner, R. Bowden, and M. Roughan. The internet topology zoo. *IEEE Journal on Selected Areas in Communications*, 29(9):1765–1775, 2011.

[13] B. Koley. The zero touch network. https://research.google.com/pubs/pub45687.html, 2016.

[14] S. Malik and L. Zhang. Boolean satisfiability from theoretical hardness to practical success. *Communications of the ACM*, 52(8):76–82, 2009.

[15] OpenConfig. Vendor-neutral, model-driven network management designed by users. http://openconfig.net. Online; accessed 22 January 2018.

[16] Palo Alto Networks. Palo Alto Networks next-generation firewalls. https://www.paloaltonetworks.com/. Online; accessed 22 January 2018.

[17] A. Panda, O. Lahav, K. J. Argyraki, M. Sagiv, and S. Shenker. Verifying reachability in networks with mutable datapaths. In *Proc. of USENIX NSDI*, 2017.

[18] R. Potharaju and N. Jain. Demystifying the dark side of the middle: a field study of middlebox failures in datacenters. In *Proc. of ACM IMC*, 2013.

[19] C. Prakash, J. Lee, Y. Turner, J.-M. Kang, A. Akella, S. Banerjee, C. Clark, Y. Ma, P. Sharma, and Y. Zhang. Pga: Using graphs to express and automatically reconcile network policies. *ACM SIGCOMM Computer Communication Review*, 2015.

[20] G. N. Purdy. Linux iptables-pocket reference: firewalls. *NAT and accounting*, 2004.

[21] M. Raynal and M. Singhal. Logical time: Capturing causality in distributed systems. *Computer*, 29(2):49–56, 1996.

[22] A. Ribeiro and H. Pereira. L7 classification and policing in the pfsense platform. In *21st International Teletraffic Congress (ITC 21), Paris, France*, 2009.

[23] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar. Making middleboxes someone else's problem: network processing as a cloud service. In *Proc. of ACM SIGCOMM*, 2012.

[24] R. Stoenescu, M. Popovici, L. Negreanu, and C. Raiciu. Symnet: Scalable symbolic execution for modern networks. In *Proc. of ACM SIGCOMM*, 2016.

[25] B. Tschaen, Y. Zhang, T. Benson, S. Banerjee, J. Lee, and J.-M. Kang. Sfc-checker: Checking the correct forwarding behavior of service function chaining. In *Proc. of IEEE NFV-SDN*, 2016.

[26] W. Wu and Y. Zhang. Network function modeling and its applications. *IEEE Internet Computing*, (4):82–86, 2017.

[27] W. Wu, Y. Zhang, and S. Banerjee. Automatic synthesis of nf models by program analysis. In *Proc. of ACM HotNets*, 2016.

[28] H. Yang and S. S. Lam. Real-time verification of network properties using atomic predicates. *IEEE/ACM Transactions on Networking*, 2016.