Model Checking for Dynamic Datapaths

Paper #82, 14 pages

ABSTRACT

We explore how to verify useful properties about networks that include “dynamic” elements, whose state and functionality may depend on previously observed traffic, e.g., caches, WAN optimizers, firewalls, and DPI boxes. We present the design and implementation of a tool that takes as input a network specification and verifies properties such as “traffic from host A will never reach host B directly or indirectly (e.g., through caching)”; or “traffic from A to B will always pass through a given middlebox (e.g., firewall or transcoder).” Our tool leverages recent advances in model checking. The challenge lies in scaling model checking with network size and complexity, and we address this by (a) modeling only globally visible middlebox behavior and (b) defining and focusing on “rest of network oblivious” (RONO) properties — properties that hold for a given traffic class independently from the rest of the network state. We have implemented our approach and can verify realistic invariants on very large networks containing 30,000 middleboxes in 2 to 5 minutes.

1. INTRODUCTION

Verification of system-wide correctness has long been important in other areas of computer science, but has only recently received much attention in networking. Several recent efforts [31,32,39] have, for the first time, provided the ability to analyze networks for global correctness. They consider networks comprised of switches and routers whose forwarding behavior is described in term of \(<\text{match, action}>\) pairs, where actions are restricted to forwarding actions or simple packet header transformations, and can detect the presence of loops, black holes, and other anomalies.

While a tremendous step forward in assuring global network correctness, these approaches are limited to network elements that perform simple IP forwarding and header rewriting based on tables handed down from the control plane. In particular, they cannot handle elements whose forwarding behavior can be changed by packet arrivals; we refer to such elements as having \textit{dynamic datapaths}, and examples include caches, stateful firewalls, WAN optimizers, DPI (deep-packet inspection) boxes and load-balancers.

One might think these are anomalous cases that can be handled as special cases, but recent studies [56] indicate that today’s networks have as many middleboxes (most of which have dynamic datapaths) as switches or routers (i.e., \# middleboxes \(\approx\) \# switches \(\approx\) \# routers). Thus, one can be sure of network correctness only if we have analyzed the behavior of these dynamic datapaths.

This is our focus in this paper. However, since these dynamic datapaths typically are not engaged in routing (i.e., making nontrivial choices about where to forward packets) but typically are deciding \textit{whether} to forward packets, our analysis will focus on three kinds of global isolation properties (we can also verify combinations of these):

- Packets from node A can (or cannot) reach network point K, where K is a host or middlebox.
- Data from node A can/cannot reach node B (even if this data is cached at a web proxy, etc.).
- Data X can/cannot reach node B (i.e., packet containing malicious data does not reach node B).

While even routers that support various IP options have dynamic datapaths — in particular, the loose source routing option bases forwarding decisions on the packet’s instructions, rather than on the destination header — for convenience in this paper we mainly focus on middleboxes and treat routers and switches as having “static” datapaths (i.e., forwarding behavior is only changed by the control plane).

To analyze whether a global invariant holds in a particular network, we treat each dynamic datapath as a “subroutine” and the network as a whole as a program. The routers and switches provide the glue that connects these procedures (and, in this paper, we assume that this glue behavior has already been verified).

Invariants specify constraints on dataflow within this program. We use symbolic model checking to determine if the specified invariants hold. So far, this is a straightforward application of standard programming language techniques to networks. However, as stated, this approach would fail to scale: middlebox code is extremely complicated, and checking even simple invariants in modest-sized networks would be intractable. Thus, the bulk of this paper is about leveraging what we know about networks to scale this approach to large networks. This involves three different aspects:

- Simplifying middlebox models, focusing only on those aspects that are relevant to the invariants.
Arguments
Node e
Address a
Node e
Node e
Packet p
Node e
Node e
Packet p
Node e
Packet p
Event R or S

<table>
<thead>
<tr>
<th>Function</th>
<th>Arguments</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>hostHasAddr</td>
<td>Node e, Address a</td>
<td>True if e is assigned address a.</td>
</tr>
<tr>
<td>send</td>
<td>Node e₁, Node e₂</td>
<td>True if node e₁ sends packet p to node e₂.</td>
</tr>
<tr>
<td>recv</td>
<td>Node e₁, Node e₂</td>
<td>True if node e₂ receives packet p from node e₁.</td>
</tr>
<tr>
<td>etime</td>
<td>Node e, Packet p</td>
<td>Returns an integer representing logical time at which node e received (R) or sent (S) packet p. Returns 0 for events that did not occur.</td>
</tr>
</tbody>
</table>

Table 1: Basic network functions.

- Simplifying network models, focusing only on the sequence of middleboxes traversed by packets.
- Identifying classes of networks where the invariants can be verified by modeling only a portion of the network.

We have implemented our approach and can verify realistic invariants on very large networks containing 30,000 middleboxes in 2 to 5 minutes.

2. BACKGROUND

We start by presenting some background on model checking, middlebox routing and network verification, and delay a more complete discussion of related work to §3.

Model checking [13, 29] is a common technique for verifying properties of programs. Initial work on model checking focused on modeling systems as non-deterministic automata (Kripke structures [37]) and proving properties on reachable states. More recently the literature has focused on symbolic model checking [40] where solvers can explore sets of states simultaneously and thus reduce the size of the search space, improving scalability. In this paper we use symbolic model checking to prove properties on networks and discuss methods for scaling these checks.

When model checking an application one must generate an equivalent (in that it captures all of an application’s relevant behavior) model that can be checked by a solver; in our case this means generating a logical model for the entire network. A logical model here is a set of variables — which together define a set of states — and a set of constraints on the values these variables can take. The constraints serve a similar purpose to transitions in models for explicit model checking. Given this logical model a solver finds an assignment (i.e., a value for each variable) which satisfies the model constraints or proves that no such assignment exists.

In this work we express our logical models in first-order logic and make use of quantifiers and uninterpreted functions. We solve our logical model using Z3 [18], an SMT [6] solver. SMT solvers can solve models expressed in first-order logic enriched with additional theories (for instance theories about uninterpreted functions, lists, dictionaries, etc.) The use of first-order logic allows the specification of a richer set of formulas (for instance formulas with universal quantifiers) than boolean logic which can be solved using SAT solvers. Despite this additional richness SMT solvers have time complexity and performance comparable to boolean SAT solvers.

Our work also builds on the existing literature on network verification, in particular on Header Space Analysis (HSA [32]) and VeriFlow [34]. We use transfer functions produced by these techniques to model the portions of the network with purely static datapaths. These tools however do not deal with the dynamic datapath elements we consider.

3. APPROACH

We now present our verification approach: we give an overview ([3.1], explain our middlebox ([3.2] and network ([3.3]) models, and describe how we use them to verify network invariants ([3.4]). Finally, in we present a few limitations of our approach ([3.5].

3.1 Overview

Our starting point is classic model checking: in our context, the network is the program, while the middleboxes are procedures that map input packets to output packets and maintain dynamic state. To prove that a network invariant holds, we need to prove that there exists no feasible state of the network model that violates the invariant.

The main challenge is scalability: Accurate models of large stateful systems typically have too many states to check with completeness, and networks are no exception. We address this in three ways:

(i) We reduce the number of states as far as possible by abstracting away internal middlebox processes; we model only externally visible behavior that is relevant to the invariants we want to verify. But that is not enough: real networks can include hundreds of middleboxes, and however simple our middlebox models, combining hundreds of them still results
in an intractable aggregate model.

(ii) We assume that packet trajectories are controlled by service chaining policies. That is, there is network policy that given a packet header specifies the series of middleboxes through which that packet should pass. There are several techniques for implementing this traversal order (for instance SIMPLE [50] and FlowTags [22]), and in what follows we assume that the ordering itself is correctly implemented by the dataplane. What we focus on is whether packets following such an order respect various network-wide invariants. This allows us to ignore routing policies and treat the network as islands of dynamic components connected by transfer functions (as defined by header-space analysis).

(iii) Instead of using a precise model for the entire network, we use precise models for a small fraction of the network, while making conservative simplifying assumptions about the rest. This approach leverages the observation that in practice, many useful global network invariants can be verified locally, i.e., by precisely modeling a subgraph of the network and over- or under-approximating the behavior of the rest of the network.

3.2 Middlebox Modeling

We model each middlebox as a set of constraints on input/output packets and internal state. A packet is a set of header fields and pseudo-fields, as well as a payload; constraints are expressed with the help of simple functions like send, recv, and etime, the latter specifying the time when a node sent or received a packet (Table 1). We present our network models in greater detail in Appendix A.

A characteristic example is the model for a stateful firewall, partly shown in Figure 1. The first constraint says that: any packet received by the firewall (f), whose source and destination addresses are not blocked by the firewall’s ACL (A), will cause the firewall to cache the packet’s addresses and port numbers (a, x, b, y). The second constraint says that, to forward a packet (p), the firewall must have previously cached its addresses and port numbers (p.src, p.src_port, p.dest, p.dest_port). The two constraints together specify that any packet allowed by the ACL will go through and will also punch a hole, allowing future traffic from the same bidirectional flow to pass in both directions.

Another characteristic example is the model for a web proxy, partly shown in Figure 2. The first constraint accounts for web requests: to forward a request p, the proxy must have previously received some other request p2, that had the same origin and destination as p. In other words, the proxy does not generate its own requests. The second constraint similarly accounts for responses from the cache.

Our main challenge is making the models rich enough to verify our target invariants, yet simple enough that verification scales to realistically-sized networks; we address this by modeling only externally visible behavior that is relevant to our target properties. For example, Figure 3 shows part of a model for a web optimizer. The two constraints specify that (i) the optimizer performs lossless compression and (ii) that: for each received packet, it will send a packet with the same header and compressed payload. That is, the model does not capture the output values of the optimizer’s compression algorithm, only the fact that it results in payload transformation. We follow the same approach with encryption and transcoding devices.

This level of abstraction results in models with relatively few transitions, yet sufficiently rich, given that our goal is to verify global network invariants, not analyze local behavior. We model compression, encryption and transcoding devices, not to analyze their algorithms, but because we need to check whether they interfere with global invariants (e.g., by changing a packet’s payload thereby enabling it to bypass a DPI box that would otherwise block it). For this, it is sufficient to capture whether and when each of them transforms the packet payload.

The resulting verification is complete (given sufficient time, it will find all invariant violations), but it may, theoretically, report a false violation. For example, consider a web optimizer followed by a DPI box which drops packets based on their payload. Suppose that the optimizer’s compression algorithm happens to leave a particular payload value v uncompressed; and it just so happens that the DPI box drops all packets with payload value v. Unless we explicitly include
this exception in our optimizer model, we will incorrectly conclude that it is possible for a packet with (original) payload value $v$ to successfully traverse the DPI box.

In our current work, we manually wrote the logical statements that comprise our models (Appendix A) based on our understanding of what these classes of middleboxes do. In practice we envision that the models themselves can either be generated by the customer (network operator) or by the vendor. Network operators already provide specifications (albeit not necessarily as formally verifiable models) for middleboxes in RFPs, and these specification are already restricted to specifying externally visible behavior. Operators can therefore write these models as a part of the RFP process and use them to both verify middlebox behavior (either using techniques from [57] or by converting the model to a set of runtime constraints against which actual middlebox behavior is compared) and analyze the impact of deploying these middleboxes in their network using our approach. Moreover, one can enforce the model by by only allowing those externally visible actions (such as a packet transmission) that fit within the constraints of the model.

In addition, prior work [35] has looked at techniques for automatically generating logical models from source code with the use of symbolic execution. Tools like KLEE [9], which builds on these techniques, can be used to extract models from system code. The extracted models are however extremely detailed (they completely model the state space of the program) and tractable analysis requires abstracting details away from these models. Some progress has been made in automated abstraction [12] at present abstraction is largely performed by humans, experts who both have semantic information about the system and about the properties under test [57].

More generally, we envision that the middlebox and NFV (network function virtualization) ecosystem will evolve so that code is always accompanied by models (either generated by the customer RFPs or the vendor). The network operator can then use these models to test the impact of adding new network functionality. This is functionally similar to the mechanisms provided by proof-carrying code [44] wherein executable binaries were accompanied by simple verifiable proofs that could be used to analyze the effects of running the executable.

### 3.3 Network Modeling

While modeling the network we assume the use of service-chaining i.e., we assume network policies that specify the series of dynamic elements (e.g., middleboxes, LSRR supporting IP routers, etc.) traversed by each packet. Based on this assumption, we divide the network into islands of dynamic components connected by static elements (e.g., switches and routers). For the static portions of the network we use HSA-like techniques to compute transfer functions (based on the chaining policy) that map a packet’s input header to an output header and output node pair. As an example, consider the transfer function shown in Figure 4 and the corresponding constraints. The first constraint specifies that packets with destination address $(p\, .\, dest) = ip_0$ and $p\, .\, port \neq 80$ are sent to host $A$ while the second one states that packets with destination address $ip_1$ are sent to host $B$. Taken together these constraints are equivalent to the transfer function $T_r$.

For any part of the network we can then model an “aggregate middlebox”, whose constraints are the conjunction of constraints for dynamic elements appearing within the subgraph and transfer functions connecting these elements. We use the term internal traffic to refer to packets that only traverse elements within the subgraph and external traffic to refer to the rest. Figure 5 shows an example of a simple subgraph and the corresponding aggregate constraints.

We also add a small set of network-wide constraints to model correct network operation: packets do not appear between network devices, a packet cannot be received before it is sent, nodes do not assign the same source and destination address to packets, etc. (Appendix A).

In terms of traffic policies, we assume that switches may perform any type of static header-based forwarding; moreover, the network operator may configure middleboxes to classify and tag packets and/or act on previously attached tags [22]. The latter makes it feasible to guide a specific
traffic class through a particular subgraph of middleboxes, e.g.,
document a DPI box to tag all packets whose payload
matches some predicate as “suspect,” then configure an IDS
box to forward all packets with this tag to a scrubbing box.
These traffic policies determine which invariants are
meaningful to verify. For instance, in the previous example,
the operator’s intent is to guide all packets whose payload matches
the given predicate to the scrubbing box; this invariant may
be violated if a web optimizer or encryption device transforms
packet payload before it reaches the DPI box. It makes
sense, then, to ask questions of the form: is it feasible for a
packet that matches a given predicate to reach a given net-
work point?

3.4 Invariants Verification

In this paper we consider the following types of invariants:
• Source isolation:

\[ \exists e: \text{send}(e, B, p) \land p.\text{src} = A. \]

Packets sent by source node \( A \) cannot reach node \( B \) (more
precisely, there exists no node \( X \) that will send such a packet
to \( B \)).

• Origin isolation:

\[ \exists e: \text{send}(e, B, p) \land p.\text{origin} = A. \]

Any data that originates at node \( A \) cannot reach node \( B \).
E.g., end-host \( B \) can never access content from web server
\( A \), directly or indirectly through a web proxy.

• Data isolation:

\[ \exists e: \text{send}(e, B, p) \land f(p.\text{orig}_\text{body}). \]

Any data that satisfies predicate \( f \) (e.g., belongs to a certain
application) cannot reach node \( B \). This is easily extended
to ensure middlebox traversal, i.e., that certain traffic must
have traversed middlebox \( M \) at some point before reaching
node \( B \).

To prove an invariant \( I \), the (conceptually) straightforward
approach would be to precisely model the entire network
as one aggregate middlebox with constraints \( C_g \), and to
use an SMT solver to find a packet \( p : C_g \land \neg I \), i.e., a packet
that satisfies both the network constraints and the inverse of
the invariant. E.g., to prove an origin-isolation invariant as
stated above, we need to look for a packet with origin \( A \) that
can reach node \( B \). If the SMT solver cannot find any such
packet, that constitutes proof that the origin-invariant holds;
if it does, that constitutes proof that the origin-invariant is
violated.

However, precisely modeling the entire network may not
scale to networks with hundreds of middleboxes: even if
we use the simplest middlebox models that capture relevant
behavior, computing the conjunction of the constraints
of hundreds of models would result in an intractable aggregate
model.

Instead, our algorithm (which we refer to as the RONO
algorithm) tries to prove global network invariants by using
precise models only for small subgraphs, while making conser-
ervative simplifying assumptions about the rest of the net-
work: To verify an invariant \( I \) that involves nodes \( A \) and \( B \),
we model the subgraph that connects these two nodes as one
aggregate middlebox with constraints \( C_l \). Then: (i) we try to
prove that \( I \) holds, assuming arbitrary external traffic; (ii)
if this fails, we try to prove that \( I \) is violated, assuming zero
external traffic; (iii) only if this fails as well, do we attempt
verification using a precise model for the entire network.

Figure 6 illustrates the three possible outcomes of the al-
gorithm: In all cases, the target invariant is origin-isolation
(data from \( A \) cannot reach \( B \)). In \([6(a)]\) the invariant holds
(the firewall \( F \) prevents \( A \) from sending packets to \( B \)), and
the algorithm proves it in step (i). In \([6(b)]\) the invariant is
violated (the firewall \( F \) allows traffic from \( A \) to \( B \)), and
the algorithm proves the violation in step (ii). In \([6(c)]\) the
invariant is violated, but we cannot prove it without precisely
modeling the entire network: if \( C \) can forward traffic from \( A \)
(sent through \( W \)) to \( B \) then the invariant is violated, which
means we cannot prove it always holds; if \( C \) sends no traf-
fic, the invariant holds, which means that we cannot be sure
it is violated without considering constraints describing \( C \)’s
behavior. We present our algorithm in greater detail in Ap-
pendix B.

In summary, the RONO algorithm can reduce verification
complexity for two classes of invariants: (i) those that hold
for arbitrary external traffic and (ii) those that are violated
for zero external traffic. These invariants are oblivious to
the rest of the network and we refer to them as “rest of net-
work oblivious” (RONO) invariants. We can therefore efficiently verify RONO invariants, but depending on network size, can also handle other invariants.

In certain cases, it is possible to make an invariant be oblivious to the rest of the network by slightly modifying the network. For instance, in Figure 6(b) if the proxy filters requests from A to B (equivalently, swapping the proxy and the firewall) would make the origin-isolation invariant hold for arbitrary external traffic, and our algorithm would prove it in step (i). We currently have a simple technique that checks whether adding static filtering rules makes an invariant hold for arbitrary external traffic. The general question of how to transform a network specification to make it verification-friendly is left to future work.

3.5 Limitations

Our current approach has a few limitations. Like most other symbolic model-checking systems we are fundamentally limited to verifying safety properties. For example this limitation implies we cannot verify properties such as “eventually there will be no loops between A and B” or that eventually a middlebox learns the identity of all correct hosts in the network, etc. We also cannot verify statements about network properties that usually hold but may be violated during transient errors (for instance an invariant of the form “all packets from A pass through rate limiter R” but this might not be sufficient in all cases.

Our technique is scalable in cases where the user of our tool can identify a subgraph within which an invariant can be verified. In the future we plan on extending our analysis with techniques like model interpolation \[41\] to automate discovery of this subgraph.

4. IMPLEMENTATION

We implemented a simple verification tool based on the algorithm and models presented in \[3\]. Our implementation is based on Python and makes use of Z3 \[18\] an open source SMT solver. Models are implemented as a straightforward translation of the logical models shown in Appendix A into code (using the Z3 API). Our implementation, which includes models for 17 types of middleboxes, is 1726 lines of code. The largest model is under 150 lines of code. Tests and benchmarks are a further 2700 lines of code; and largely define the transfer functions between dynamic elements.

We define middlebox models (which we expect change infrequently and are largely reused across networks and perhaps even implementations) using first-order formulas. We specify transfer functions as a list of predicate, output header, output node tuples. Networks are specified as a list of middleboxes and transfer functions describing how they are connected. From this input we produce a model that contains functions which can be queried to determine whether particular invariants are correctly enforced.

For evaluation we set the following parameters for Z3, which are listed below:

- We turned off Z3’s auto-configuration setting (to reduce any effects due to auto-configuration, etc.).
- We enabled model based quantifier instantiation (MBQI). MBQI is a solving strategy for formulas with universal quantifiers, which we use extensively in expressing our model.
- We increased the quantifier instantiation depth and the maximum number of MBQI iterations. This allows the solver to verify models (or alternately prove that no satisfiable assignment existed) in cases where an individual path contains several elements.
- We increased the solver timeout so we could evaluate larger models.

1 as measured using David A. Wheeler’s sloccount
By default Z3 seeds the random number generator with 0. Each iteration we set the seed value to a number read from the system random number generator. SMT solvers make extensive use of random numbers and hence the seed can dramatically affect performance.

We experimented with setting other solver parameters and found that the choice of parameters for certain networks can improve verification time by up to an order of magnitude. These parameters however did not apply to all networks and hence we do not use them for our evaluation.

5. EVALUATION

In this section, we answer the question: are there realistic scenarios where our approach can help uncover unintended network behavior (§5.1)? We also examine how our approach scales as a function of different network characteristics (§5.2).

5.1 Use Cases

We present three realistic scenarios where our proposal would be useful. In all of them, each network device individually behaves as expected, yet their interaction causes the network to violate the global behavior intended by the operator. We show verification times for the subgraph in isolation (i.e., a case where we completely model the network under test) for each of these realistic scenarios in Table 2.

The violations presented here appear simple when considered in isolation and could probably be checked manually. However, invariant violations of the form we present are likely to be introduced when network operators add new middleboxes to their networks. For instance a network provider might add a proxy in order to reduce network utilization. As we show later, adding a proxy might result in the network violating an origin isolation invariant. Manual analysis to determine whether adding a middlebox causes an invariant violation would require the administrator to analyze all such combinations. The bugs introduced by faulty analysis in this case are not much more complicated than our examples, and yet have real consequences. Automating analysis reduces the possibility of human error and hence increases network reliability.

**Loose Source Record Route + Firewall.**

We start by considering a case where the network cannot be approximated. We try and prove origin-isolation such that data from host A cannot reach host B in the subgraph shown in Figure 7(a). Here L0 is an IP router supporting LSRR and F is an ACL firewall.

Routers supporting the loose source record route (LSRR) IP option allow end-hosts to specify, as a set of IP options, a list of intermediate routers that the packet should be routed through; upon receiving the packet, each of these routers sets the packet’s source address to its own and its destination address to the next intermediate router on the list. To achieve the origin-isolation invariant the operator configures the firewall F to drop all packets with source address A and destination address B.

The target invariant is violated if end-host A sends a packet addressed to B and specifies a LSRR router as an intermediate point (causing the latter to rewrite the packet’s source IP address and bypass the firewall). This may seem obvious in retrospect, yet it caused enough trouble in practice to make network operators disable support for this IP option and led router manufacturers to revise the LSRR specification.

For a network with one LSRR enabled router (where we completely model the network) our algorithm can check this invariant (and find a violation) in 0.42 seconds on average.

We can also analyze a similar invariant violation in a larger network containing several LSRR routers. In such a network any LSRR router can route traffic to any other point in the network and packets can traverse an unbounded (LSRR does not prohibit loops in the routes specified) set of paths. These networks are also not amenable to approximation: a packet can take any path through the network and hence we must completely model the entire network.

We analyze a network (Figure 8(a)) with a varying number of LSRR routers, two hosts (A and B) and a firewall F. All traffic through B must traverse the firewall F and A can send traffic to all points in the network. First we consider the case where F is only configured to drop packets with source A and destination B. Figure 8(b) shows time taken to find the invariant violation. For a network with 35 LSRR routers we can find a violation in 75.69 seconds on average.

Second, we consider the case where F is configured to drop packets from all but one of the LSRR routers to B (in addition to dropping packets from A). This policy greatly reduces the number of valid models in the search space and hence leads to increased verification time. Figure 8(c) shows...
time for checking this invariant with increasing node count. With this increased complexity it takes 2056.40 seconds (approximately 35 seconds to find violations in a network with 35 LSRR routers.

Web Optimizer + DPI Firewall.

Next, we consider a RONO invariant: we check to see if a packet sent by $A$, carrying a malicious byte sequence can reach $B$ in the subgraph depicted in Figure 7(b). In this subgraph $W$ is a web optimizer which can compress HTTP payloads—a capability offered by several state-of-the-art network appliances [1,2,55]. $D$ is a deep-packet inspection (DPI) firewall configured to drop packets whose payload matches any of a given set of predicates e.g., payloads with a byte sequence that typically indicates a known exploit [51,60].

The target invariant will be violated if a malicious packet (i.e., a packet containing a malicious byte sequence) sent by $A$ is compressed by the optimizer $W$ in which case the DPI firewall might misclassify the packet and allow it through.

First we consider the subgraph in isolation. In this case the network is completely modeled and we can find the invariant violation in 1.64 seconds on average.

We also measure time taken to verify this invariant when this subgraph is embedded in a larger network. In this case we use the RONO algorithm terminates in step (ii) after showing that the invariant is violated in the under-approximate network (i.e., a network where none of the other nodes send any traffic). When embedded in a network with 30,000 nodes (middleboxes of various kinds) analysis, using the RONO algorithm, takes 84.66 seconds on average.

Web Proxy + Firewall.

Finally we consider an invariant that is affected by the behavior of the rest of the network, but can be easily transformed to a RONO invariant. We check the invariant that end-host $A$ cannot access any data originating from the web server $B$ in the subgraph shown in Figure 7(c). The network includes a standard web proxy $P$ and an application-specific firewall $F$ that can bases its decision on the packet’s source and destination and on HTTP headers indicating that packet is being proxied for another host (in particular the X-Forwarded-For HTTP header [47]). The firewall is configured to drop web requests originating from $A$. The network also contains at least one other end-host $C$.

This is not a RONO invariant: $C$ can send a request to $B$ resulting in $P$ caching the response. This implies that when over-approximated the invariant is violated, while when under-approximated the invariant is not violated. Proving that this invariant is not RONO takes 7.28 seconds (RONO Check in Table 2). Precisely modeling the network and proving the invariant violation take 4.61 seconds on average.

However, consider the case where the network contains several other end hosts (rather than just $C$) in addition to $A$ and $B$ (Figure 9(a)). Figure 9(b) shows time taken to check the invariant with increasingly large number of nodes. We find that for less than 70 nodes it can take over 400 seconds to run verification. Proving that the network is not RONO scales better (Figure 9(c)) and we can check this property for up to a 1000 node in under 2 minutes.

Next we consider a similar RONO invariant: by swapping the positions of $F$ and $P$ in the pipeline (Figure 9(d)) one can make the invariant RONO. In this case $F$ can drop requests from $A$ before they reach $P$. The precise subgraph for this
network can be verified in 0.08 seconds.

Next we compare the time taken for verifying the invariant on precise graphs of increasing size as compared to using the RONO algorithm. Figure 9(e) shows the time taken to prove the invariant violation on a precisely modeled graph while Figure 9(f) shows time taken using the RONO algorithm. Testing a RONO invariant significantly improves the performance of both: in case of the precise graph we believe the SMT solver can use heuristic to derive the same over-approximate model that RONO uses.

At a much larger scale we find that with 30,000 nodes the RONO algorithm takes 87.47 seconds to prove invariant violation, while it takes 362.85 seconds on the precisely modeled graph.

5.2 Scalability Microbenchmarks

Next we analyze how verification scales with increasing complexity of a precisely modeled network. We analyze how the verification time is affected by: (i) increasing the number of middleboxes a packet must traverse, (ii) increasing number of parallel paths through the network and (iii) increasing the configuration complexity for a single middlebox.

First, we analyze the cost of increasing the number of middleboxes traversed. Figure 10(a) shows our setup: we test for the invariant $A$ is isolated from $B$. $F_0$, $F_1$, ..., $F_n$ are firewalls which do not block any traffic. Figure 10 shows the results of our experiment—we can prove invariant violations on paths with 20 middleboxes in under 35 seconds.

Next, we analyze the cost of increasing the number of parallel paths through a network. We again test whether a host $A$ is isolated from $B$ but in a precise network of the form shown in Figure 11(a). Here $F_0$, ..., $F_n$ are equivalent firewalls (in that traffic can go through any of them) which do not block any traffic. Figure 11(b) shows the time taken to prove an invariant violation in this setup—we can prove violations on networks with 50 equivalent firewalls in under 30 seconds.

We find that increasing the number of parallel paths scales better (in terms of time complexity) than increasing path length. When finding a model for the case with increasing path length the solver must find a model which satisfies constraints for all components while with parallel paths it needs to satisfy constraints for any one of the paths. Since in this case all paths are equivalent the search problem is simpler.

Finally, we measure the impact of increasing configuration complexity on verification performance. Middlebox configuration must be translated into a set of constraints and hence increasing configuration complexity. To measure the impact of these additional constraints we set up a graph with two hosts ($A$ and $B$) and a firewall $F$ as shown in Figure 12. We assign several addresses to each host (similar to how one might multi-homed hosts in a real network) and add ACLs to the firewall to block these multiple addresses. We explicitly remove one of the address pairs to introduce a violation and run our tool to detect invariant violations. Our results are shown in Figure 12(b) and we can find the violation in under 50 seconds for a firewall enforcing about 2500 ACL rules.

6. RELATED WORK

Our approach builds on the model checking literature which we review here to provide context. We also review and compare our approach to existing work in network verification.
6.1 Model Checking

Model Checking was initially proposed as a mechanism for hardware and protocol verification. The earliest approach to model checking, explicit state model checking, involved modeling (expressing) programs as a (possibly unbounded) abstract state machine with a set of initial states and a set of “interesting states”. Verification was performed by exhaustively exploring the set of reachable states [17] and checking if any of the interesting states were reachable.

Scaling explicit state model checking so it can verify complex specifications with large state space is a long standing challenge [15] (see [13] for a variety of proposals on how to handle this explosion in state space). Symbolic model checking [40] proposed that the state space be partitioned into a smaller number of sets (constraints on which were represented using binary decision diagrams [4]). This reduced the size of the search space and made many previously intractable problems tractable.

Symbolic model checking has since been extended to allow formulas to be represented in a variety of different logical systems including boolean logic [41], first order logic [6], temporal logic [48] and higher-order logic [53]. In the distributed algorithms community temporal logic which is checked using TLA+ [38] is widely used to verify new algorithms [11, 21, 30]. Our approach is largely independent of the logical system and solver used and is easily extended to apply to other logical systems.

Scalability concerns have also plagued symbolic model checking. Model abstraction [20, 26, 33, 57] has been previously used to allow symbolic model checking to be applied to large systems. Model abstraction relies on approximating the models to reduce the state space to be checked. This abstraction is commonly done manually by experts [57]. Our use of approximate middlebox models (§3.2) and our strategy of over-approximating and then under-approximating the overall network (§3.4) are forms of model abstraction.

Compositional reasoning [16] is an alternate approach, used for scaling symbolic model checking. A model checker using compositional reasoning divides the system under analysis into several smaller components (often the components are functions or components marked by the developer) and verifies individual components. The model checker then combines the result of verification across components to prove properties of the entire system as a whole.

Previous work on automatic model abstraction in particular CEGAR [12] and program slicing [59] iteratively run over-approximated and under-approximated models to decide whether a generated model is rich enough to verify a given invariant. The decision process used is similar to our algorithm for checking a network subgraph (Appendix B).

6.2 Network Verification

The earliest use of formal verification in networking was for proving correctness and security properties for protocols [14, 52]. Only recently has formal verification been used to analyze properties for the network control and data plane: the early work [23, 24] looked at verifying BGP configuration in wide-area networks.

Anteater [39] has previously looked at applying model checking (with models expressed in boolean logic) to network data planes. Anteater takes a snapshot of the switch FIBs across the network and translates FIB entries into a set of boolean formulas which are then used to verify invariants. Anteater assumes that data plane behavior does not affect the FIB (and in fact that data plane behavior is stateless) disallowing dynamic components. Anteaters use of abstract values for packet headers is similar to our use. The use of symbolic packet histories serves a similar purpose as pseudo-fields in our models of packets (Appendix A).

Several recent tools target the control and data plane for software defined networks. Among these NICE [10] uses model checking to verify network correctness, by analyzing the correctness of controller applications in software-defined networks. These applications are largely dependent on a single “logically centralized” controller and hence NICE cannot easily be applied to verifying the effect of dynamic datapaths which tend to be more distributed. Our approach does not address correctness for the control plane and hence is largely orthogonal to NICE. NICE leverages concolic execution [55], a technique where values from concrete program execution are combined with constraints generated using symbolic execution, to make the verification problem tractable.

Header-space analysis [31, 32] and Veriflow [34] use static analysis to verify data plane correctness. They assume static data plane elements and use a verification approach that treats

![Diagram](a) Setup

(b) Results: The error bars indicate minimum and maximum times across 50 runs.

Figure 12: The cost of adding constraints without changing the subgraph.
packet headers as a sequence of uninterpreted bits and network elements as functions mapping packet headers to new packet headers and output ports. Modeling packet headers in this manner allows these approaches to generate concrete counterexamples of packets resulting in invariant violation. Our tool can only generate abstract counterexamples, converting these to a concrete form might be impossible in some cases. Treating headers as uninterpreted bits unfortunately also means that verification scales with number of bits that need to be considered and as a result these systems are ill-suited for verifying routing policies like LSRR (where the “packet header” must include IP options which are large and potentially unbounded).

Recent work [19] has also looked at efficient verification for an individual middlebox. This work shows techniques that can be applied to dataplane code (for instance middlebox code) written in a modular pipelined fashion (using tools like Click [43]) so they can be efficiently verified using compositional reasoning. The models generated by this work are complete and cannot be used (unmodified) as inputs for our tool. Extending our tool to accept these models as input (and abstracting them) is left to future work.

Similarly, recent work has also proposed language extensions [25,46] to simplify the verification of network control planes. In general the use of language extensions, for example code annotations, to simplify verification has been widely studied [8,27,28,45]. We envision that similar techniques can be used to simplify the process of generating and abstracting middlebox models for use with out tool.

7. FUTURE WORK

We conclude by discussing possible future work.

We currently depend on a user specified subgraph to decide the approximation boundary used by the RONO algorithm. Recent efforts in the PL community have shown that interpolation [42], a technique where constraints are iteratively added to discover the smallest model required to verify an invariant, is a powerful method for automatic abstraction. The RONO algorithm provides a start for applying interpolation-like techniques to network verification. Extending the algorithm in this manner is left to future work.

In evaluating our approach we showed that the RONO algorithm can be used to efficiently check if an invariant is oblivious to the rest of the network. However, our evaluation also showed that verifying a RONO invariant on a precise model of the entire network using our simplified middlebox models is tractable, even for networks of several tens of thousands of nodes. We believe that this is a result of the SMT solver using strategies that detect and perform the same over-approximation as the RONO algorithm. SMT solvers however rely on heuristics to perform such optimization and we think their exist examples that are RONO invariant but do not lend themselves to automatic over-approximation. Finding such an example for Z3 and other solvers remains an open question.

SMT solvers have made tremendous progress in recent years [5] and this progress have been largely driven by applications: as applications gain popularity SMT solvers add solving strategies that are tuned to the application domain. Finding strategies that improve verification time for networks is left to future work.

Finally, we currently do not provide a general mechanism for transforming a network specification into an easily verifiable specification. Specifically, we believe their exists a mechanism for automatically transforming non-RONO invariants to RONO-invariants. Finding this mechanism is left to future work.

8. REFERENCES


APPENDIX

A. MODELS

Below we present some basic information about how we model the network, express invariants and show models for a few middleboxes.

A.1 Network

A packet is one of the basic types modeled by our network. In addition to packets a network model also contains a set on uninterpreted constants representing nodes (i.e., middleboxes and endhosts) and addresses. A packet is a collection of fields representing the standard packet header fields, payload and two pseudo-fields representing the unmodified payload (i.e., the original payload before compression or other transformations) and data origin. The fields were chosen because they commonly appeared in our models and in the invariants we checked, however more fields can be added by defining a new function that takes a packet as an input. For example the $x_{frr}$ function is used in Figure 2 to model the HTTP $X$-Forwarded-For field. As long as all models using the same field agree on the function name, adding a new field is trivial. Additionally the basic model defines four functions shown in Table 1. The models themselves are expressed as constraints on the set of packets that can be sent by a node.

All models also include a basic set of constraints (the final model passed to the SMT solver is the logical conjunction of all constraints) which define the behavior of the basic functions. These constraints include:

- All received packets were once sent

  $\forall e_1, e_2, p: \text{recv}(e_1, e_2, p) \implies \text{send}(e_1, e_2, p)$

- Disallow loopback

  $\forall e_1, e_2, p: \text{send}(e_1, e_2, p) \implies e_1 \neq e_2$

  $\forall e_1, e_2, p: \text{send}(e_1, e_2, p) \implies p.\text{src} \neq p.\text{dest}$

- Time goes forward

  $\forall e_1, e_2, p: \text{recv}(e_1, e_2, p) \implies e\text{time}(e_2, p, R) > e\text{time}(e_1, p, S)$
Time is not 0 for received packets
\[ ∀e_1, e_2, p : recv(e_1, e_2, p) \implies etime(e_2, p, R) > 0 \]

Time is not 0 for sent packets
\[ ∀e_1, e_2, p : send(e_1, e_2, p) \implies etime(e_1, p, S) > 0 \]

Time is 0 for packets that were not received
\[ ∀e_1, p : \neg(∃e_2 : recv(e_1, e_2, p)) \implies etime(e_2, p, R) = 0 \]

Time is 0 for unsent packets
\[ ∀e_1, p : \neg(∃e_2 : send(e_1, e_2, p)) \implies etime(e_1, p, S) = 0 \]

A.2 Invariants

To check an invariant we add constraints to the model asserting that a packet \( p_0 \) exists such that the inverse of the invariant holds for \( p_0 \). The SMT solver then tries to find a satisfiable assignment that both satisfies all the network constraints and the inverse of the invariant. If no satisfiable assignment can be found we know that the negation of our constrain must hold i.e., there exist no packet that both violates the invariant and satisfies the modeled constraints. On the other hand if the solver finds a satisfiable assignment we have found a counterexample where the invariant is violated.

Next we show the constraints added for a few invariants:
- Isolation between source \( A \) and destination \( B \)
  \[ \exists e : recv(e, B, p_0) \land p_0.\text{dest} = A \]
- Data Isolation i.e., ensuring client \( A \) cannot access data from service \( B \).
  \[ \exists e : recv(e, A, p_0) \land p_0.\text{origin} = B \]
- Prevent certain kind of data defined by predicate \( f \) from reaching \( A \).
  \[ \exists e : recv(e, A, p_0) \land f(p_0.\text{body}) \]

A.3 Component Models

Below we show models for a few different network elements.

In modeling the various elements a few constraints reoccur commonly and we abstract them out and list them first. Check whether two packets have identical headers.

\[ \text{PacketHeadersEqual}(p_1, p_2) : (p_1.\text{src} = p_2.\text{src}) \]
\[ \land (p_1.\text{dest} = p_2.\text{dest}) \]
\[ \land (p_1.\text{src}.\text{port} = p_2.\text{src}.\text{port}) \]
\[ \land (p_1.\text{dest}.\text{port} = p_2.\text{dest}.\text{port}) \]
\[ \land (p_1.\text{options} = p_2.\text{options}) \]

Make sure the node does not send a packet addressed to itself.

\[ \text{DontSendSelf}(A) : \forall e, p : send(A, e, p) \implies \neg\text{hostHasAddr}(A, p.\text{dest}) \]

Make sure the packet is not duplicated and sent to two different hosts.

\[ \text{SendOnce}(A, B, p) : \neg(\exists e : e \neq B \land send(A, e, p)) \]

Receive packet before sending it.

\[ \text{TemporalOrdering}(A, p) : etime(A, p, R) < etime(A, p, S) \]

A.3.1 End Hosts

We make minimal assumptions about end hosts (\( A \) in the next set of constraints), the constraints mainly ensure that the packet pseudo-fields (origin and original data) are correctly populated. We also ensure that the packet source field is not spoofed, we assume network operators can use existing techniques to enforce such a policy.

\[ ∀e, p : send(A, e, p) \implies \text{hostHasAddr}(A, p.\text{src}) \]
\[ ∀e, p : send(A, e, p) \implies p.\text{origin} = A \]
\[ ∀e, p : send(A, e, p) \implies p.\text{orig}_\text{body} = p.\text{body} \]

A.3.2 LSRR Routers

Routers supporting loose source routing \[49\] change destination addresses based on the set of IP options provided. To model them we first create a function \( \text{lsrr} \) that given the current set of IP options and an address returns the next destination to which a packet should be sent.

\[ \text{lsrr} : p.\text{options} \times p.\text{dest} \rightarrow \text{address} \]
\[ ∀a : \text{lsrr}(\emptyset, a) = a \]

There are two types of packets that an LSRR router (\( L \) in the constraints) can send out: packets which are merely being tunneled through it (i.e., packets which have a source and destination address that does not belong to the router) or packets which have been modified so the source address is set to the router’s source address and the destination address is computed using the \( \text{lsrr} \) function. We represent this as:

\[ ∀p_0, e_0 : \text{send}(L, e_0, p_0) \implies \]
\[ (\exists e_1 : \text{recv}(e_1, L, p_0) \land \neg\text{hostHasAddr}(L, p_0.\text{dest}) \land \text{TemporalOrdering}(L, p_0)) \]
\[ \lor (\exists e_1, p_1 : \text{recv}(e_1, L, p_1) \land \text{hostHasAddr}(L, p_1.\text{dest}) \land \text{SendOnce}(L, n, p_0) \land p_0.\text{src} = p_1.\text{dest} \land p_0.\text{dest} = \text{lsrr}(p_1.\text{options}, p_1.\text{dest}) \land \text{Copy other fields}) \]

A.3.3 Firewalls

In this subsection we focus on firewalls whose configuration (ACLs, application specific configuration or DPI configuration) defines conditions based on which packets should
be dropped (i.e., the rules define the set of packets to be
dened). Constraints for nders which instead specify the set
of packets to be allowed are speciﬁed by negating the deny
criterion.
For all kinds of nders a basic building block is the pred-
crate for when a packet should be denied deny. In case
of ACL based nders this predicate can be expressed as
deny(p) : (p.src, p.dest) ̸∈ ACLs, for DPI nders the
predicate might be deny(p) : dpi(p), etc. The predicates
might be composed either using logical conjunction or dis-
junction. Assuming the existence of a suitable deny
predicate a stateless nder (F) might be speciﬁed as simply as:
\[
\forall p, e_0 : \text{send}(F, e_0, p) \implies \exists e_1 : \text{recv}(e_1, F, p) \\
\wedge \neg(deny(p)) \\
\wedge \text{SendOnce}(F, e_0, p) \\
\wedge \text{DontSendSelf}(F, e_0, p) \\
\wedge \text{TemporalOrdering}(F, p)
\]
A stateful nder (as shown in § 3.2) requires two pre-
dicates deny_p that speciﬁes that a particular packet should
be denied and deny_f indicating that a certain set of ﬂows
should never be allowed. The rest of the behavior is similar
to what was previously shown.

B. RONO ALGORITHM

The RONO algorithm provides a decision process for de-
ciding whether an invariant is RONO. The algorithm (Al-
gorithm 1) accepts an over-approximated model (m_{path}) as
input and proceeds as follows:
(i) First we check the invariant on the over-approximate
model. If the invariant holds the algorithm decides that the
invariant is RONO and exits. An over-approximate model
has fewer constraints than a precise model. Also proof that
an invariant holds means that a counter-example (valid as-
signment cannot be found). Adding more constraints to the
model cannot change this outcome.
(ii) Next we build an under-approximate model from the
over-approximate model by asserting that none of the ap-
proximate nodes can send packets. We check if the invariant
is violated in the under-approximate model. If the invariant
is violated in the under-approximate model the algorithm de-
cides that the invariant is RONO and exits. A violation in the
under-approximate model implies a violation in the precise
network (which allows strictly more nodes to send packets).
(iii) Finally if neither holds we declare that the invariant
is not RONO.

Similar to mechanisms like CEGAR [12] our algorithm
can be extended to nd the smallest over-approximate model
where an invariant is RONO, however this comes at the cost of
veriﬁcation time.

| Algorithm 1: RONO Algorithm: Determine if an invari-
<table>
<thead>
<tr>
<th>ant is rest of network oblivious (RONO)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data: m_{path} The model for the subgraph</td>
</tr>
<tr>
<td>elements set of elements in path</td>
</tr>
<tr>
<td>Result: judgement: Is this path isolated or not</td>
</tr>
<tr>
<td>begin</td>
</tr>
<tr>
<td>// Check if invariant holds in over-approximated network</td>
</tr>
<tr>
<td>d_{local} ← m_{path}.check()</td>
</tr>
<tr>
<td>if d_{local} == not_violated then</td>
</tr>
<tr>
<td>// Result unaffected by additional constraints</td>
</tr>
<tr>
<td>judgement ← RONO</td>
</tr>
<tr>
<td>else if d_{local} == unknown then</td>
</tr>
<tr>
<td>judgement ← unknown</td>
</tr>
<tr>
<td>else</td>
</tr>
<tr>
<td>// Check if the current model already proves the under-approximated case</td>
</tr>
<tr>
<td>participants ← m_{path}.model().nodes</td>
</tr>
<tr>
<td>if participants - elements ≠ ∅ then</td>
</tr>
<tr>
<td>// Must explicitly check under-approximated network</td>
</tr>
<tr>
<td>// Add constraints to under-approximate network</td>
</tr>
<tr>
<td>m_{path}.add(\forall e, n, p : send(e, n, p) ⇒ e ∈ elements)</td>
</tr>
<tr>
<td>d'<em>{local} ← m</em>{path}.check()</td>
</tr>
<tr>
<td>if d'<em>{local} == d</em>{local} then</td>
</tr>
<tr>
<td>// Invariant is violated even in an underapproximated network</td>
</tr>
<tr>
<td>judgement ← RONO</td>
</tr>
<tr>
<td>else</td>
</tr>
<tr>
<td>// Invariant is not violated in an underapproximated network. We must consider the entire network.</td>
</tr>
<tr>
<td>judgement ← not_RONO</td>
</tr>
<tr>
<td>return</td>
</tr>
<tr>
<td>else</td>
</tr>
<tr>
<td>// The original proof holds for the underapproximated case.</td>
</tr>
<tr>
<td>judgement ← RONO</td>
</tr>
</tbody>
</table>